REPRODUCIBLE CODE

BRITISH ECOLOGICAL SOCIETY

Contents

BRITISH ECOLOGICAL SOCIETY

Editors

Natalie Cooper, Natural History Museum, UK and **Pen-Yuan Hsing**, Durham University, UK

Contributors

Mike Croucher, University of Sheffield, UK

Laura Graham, University of Southampton, UK

Tamora James, University of Sheffield, UK

Anna Krystalli, University of Sheffield, UK

François Michonneau, University of Florida, USA

Introduction

Natalie Cooper, Natural History Museum, UK

Pen-Yuan Hsing, Durham University, UK

The way we do science is changing — data are getting bigger, analyses are getting more complex, and governments, funding agencies and the scientific method itself demand more transparency and accountability in research. One way to deal with these changes is to make our research more reproducible, especially our code.

Although most of us now write code to perform our analyses, it is often not very reproducible. We have all come back to a piece of work we have not looked at for a while and had no idea what our code was doing or which of the many "final_analysis" scripts truly was the final analysis! Unfortunately, the number of tools for reproducibility and all the jargon can leave new users feeling overwhelmed, with no idea how to start making their code more reproducible. So, we have put together this guide to help.

A Guide to Reproducible Code covers all the basic tools and information you will need to start making your code more reproducible. We focus on R and Python, but many of the tips apply to any programming language. **Anna Krystalli** introduces some ways to organise files on your computer and to document your workflows. **Laura Graham** writes about how to make your code more reproducible and readable. **François Michonneau** explains how to write reproducible reports. **Tamora James** breaks down the basics of version control. Finally, **Mike Croucher** describes how to archive your code. We have also included a selection of helpful tips from other scientists.

True reproducibility is really hard. But do not let this put you off. We would not expect anyone to follow all of the advice in this booklet at once. Instead, challenge yourself to add one more aspect to each of your projects. Remember, partially reproducible research is much better than completely non-reproducible research.
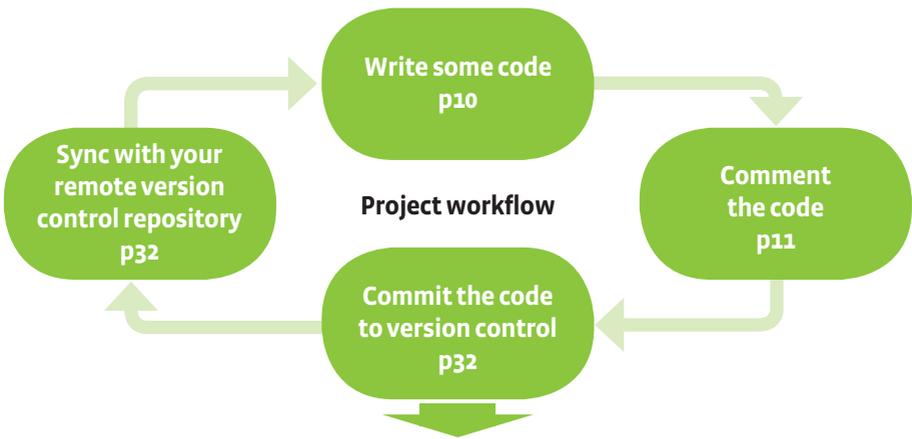
Good luck!

A simple reproducible project workflow

Before you start

- Choose a project folder structure. p6
- Choose a file naming system. p7
- Choose a coding style. p10
- Install and set up your version control software. p29
- Set up an online version control account. p29

First steps

- Create the project folder and subfolders. p8
- Add a README file describing the project. p6
- Create a version control repository for the project using git or similar. p28
- Connect the version control repository to an online remote repository. p32
- Add a LICENSE file. p36
- Create a new reproducible report for the project. p20

Project workflow

Write some code p10

Comment the code p11

Commit the code to version control p32

Sync with your remote version control repository p32

Preparing for publication

- Record the versions of all the packages and software you used p18
- Update the README to contain details of the project workflow, package versions etc.
- Get a friend or colleague to check all your documentation makes sense and nothing is missing
- Add additional documentation if needed. This is vital if you are writing a package for others to use
- Add a LICENSE file if you have not already added one p36
- Make your online remote repository public if it was private p28
- Archive the code and get a DOI so it can be cited p36
- Remember to archive and document your data too! (***BES Guide to Data Management***)
- Bask in the glory of you reproducible masterpiece and watch the citations come in!

Organising projects for reproducibility

Anna Krystalli, University of Sheffield, UK

The repeatable, reproducible analysis workflow

The fundamental idea behind a robust, reproducible analysis is a clean, repeatable script-based workflow (i.e. the sequence of tasks from the start to the end of a project) that links raw data through to clean data and to final analysis outputs. Most analyses will be re-run many times before they are finished (and perhaps a number of times more throughout the review process), so the smoother and more automated the workflow, the easier, faster and more robust the process of repeating it will be.

Principles of a good analysis workflow

- Start your analysis from copies of your raw data.
- Any cleaning, merging, transforming, etc. of data should be done in scripts, not manually.
- Split your workflow (scripts) into logical thematic units. For example, you might separate your code into scripts that (i) load, merge and clean data, (ii) analyse data, and (iii) produce outputs like figures and tables.
- Eliminate code duplication by packaging up useful code into custom functions (see **Programming**). Make sure to comment your functions thoroughly, explaining their expected inputs and outputs, and what they are doing and why.
- Document your code and data as comments in your scripts or by producing separate documentation (see **Programming** and **Reproducible reports**).
- Any intermediary outputs generated by your workflow should be kept separate from raw data.

Organising and documenting workflows

The simplest and most effective way of documenting your workflow – its inputs and outputs – is through good file system organisation, and informative, consistent naming of materials associated with your analysis. The name and location of files should be as informative as possible on what a file contains, why it exists, and how it relates to other files in the project. These principles extend to all files in your project (not just scripts) and are also intimately linked to good research data management, so check the **BES Guide to Data Management in Ecology and Evolution**[1] for more information.

---

[1] British Ecological Society: *A Guide to Data Management in Ecology and Evolution.*

Organising projects for reproducibility

File system structure

It is best to keep all files associated with a particular project in a single root directory. RStudio projects offer a great way to keep everything together in a self-contained and portable (i.e. so they can be moved from computer to computer) manner, allowing internal pathways to data and other scripts to remain valid even when shared or moved (see **Programming**). An exception can be code associated with a number of projects that may live in its own directory (although you could consider making this into a package, see r-pkgs.had.co.nz/ for advice on making packages in R).

There is no single best way to organise a file system. The key is to make sure that the structure of directories and location of files are consistent, informative and works for you.

Here is an example of a basic project directory structure:

- The `data` folder contains all input data (and metadata) used in the analysis.
- The `doc` folder contains the manuscript.
- The `figs` directory contains figures generated by the analysis.
- The `output` folder contains any type of intermediate or output files (e.g. simulation outputs, models, processed datasets, etc.). You might separate this and also have a `cleaned-data` folder.
- The R directory contains R scripts with function definitions.
- The `reports` folder contains RMarkdown files that document the analysis or report on results.

Organising projects for reproducibility

- The scripts that actually do things are stored in the root directory, but if your project has many scripts, you might want to organise them in a directory of their own.

Never ever touch raw data. Store them permanently, and use scripts to produce derived, clean datasets for analyses.
– Francisco Rodríguez-Sánchez, Estación Biológica de Doñana (CSIC)

Name
▶ 📁 data
▶ 📁 doc
▶ 📁 figs
▶ 📁 output
▶ 📁 R
▶ 📁 reports
r 01_download-data.R
r 02_clean-data.R
r 03_exploratory-analysis.R
r 04_fit-models.R
r 05_generate-figures.R
R analysis-example.Rproj
README.md

Fig 1. Example file structure of a simple analysis project
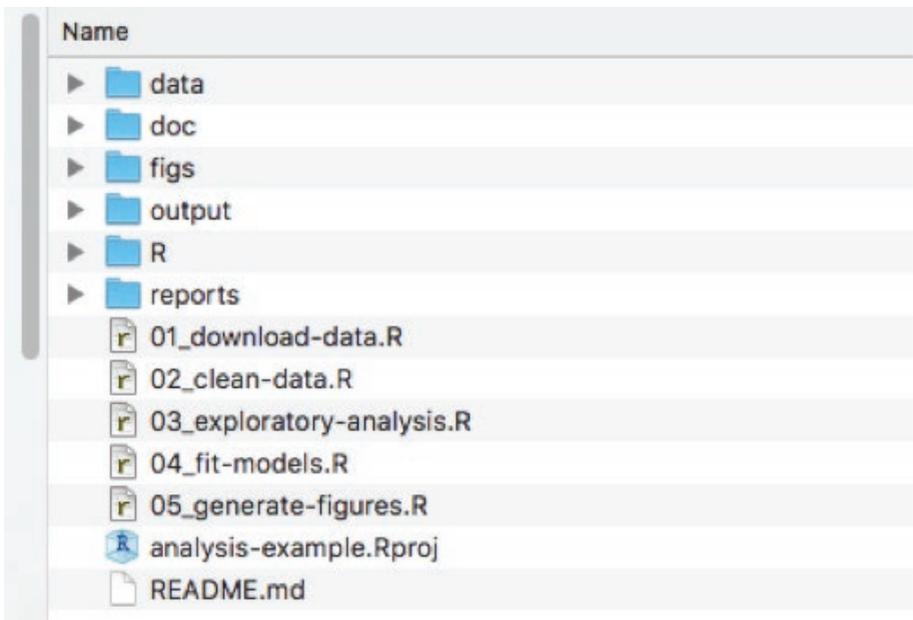
For every project, keep a text document in which you record what you did and any rationale, concerns, ideas, etc., that go with it. If you are working on the command line (e.g. using QIIME), copy and paste the exact commands you used into this document. It will allow you to retrace your steps and train of thought at a later date. – Sarah Johnson, Cardiff University

Organising projects for reproducibility

Folder structure

- Be consistent – when developing a naming scheme for folders it is important that once you have decided on a method, you stick to it. If you can, try to agree on a naming scheme at the start of your research project.

- Structure folders hierarchically. Start with a limited number of folders for the broader topics, and create more specific folders within these as and when they are required.

- Include a README file to describe the project and provide some basic orientation around project files. Good naming of files should take care of the rest, and remember to describe the naming scheme in the README file.

- For files that cannot be commented or described easily by name, include file specific READMEs in your folders that describe metadata such as the contents of the file, sources of the contents, links to relevant papers or data repositories, the units of the measures included in data columns, etc.

- If you are making your work public for others to reuse, make sure you include an appropriate license (see **Archiving**).

- Quarantine inputs and materials given to you by others (i.e. keep it in a separate folder). Rename and repurpose into your own file system, recording what has been done. Ensure provenance of inputs is tracked.

- Keep track of ideas, discussions and decisions about analysis. An info folder where you store important emails or idea drafts could work. Otherwise, many online version control (see **Version control**) hosting services (such as, but not limited to, GitHub) have issues management which provides a lot of useful functionality for this purpose.

- Separate ongoing and completed work: as you start to create lots of folders and files, it is a good idea to think about separating older documents from those you are currently working on.

Informative, consistent naming

Good naming extends to all files, folders and even objects in your analysis (see **Programming**) and serves to make the contents and relationships among elements of your analysis understandable, searchable and organised in a logical fashion. For example, informative, consistent naming can allow you to search for data generated during a specific time period, data generated by a particular script, analysis outputs generated using specific values of a parameter and objects

Organising projects for reproducibility

generated by a specific function.

> **Examples of bad vs. good filenames**
>
> | BAD | BETTER |
> |---|---|
> | `01.R` | `01_download-data.R` |
> | `abc.R` | `02_clean-data_functions.R` |
> | `fig1.png` | `fig1_scatterplot-bodymass-v-brainmass.png` |
> | `IUCN's metadata.txt` | `2016-12-01_IUCN-reptile_shapefile_metadata.txt` |

Key principles of good file naming – make sure they are:

Machine readable

- Avoid spaces, punctuation, accented characters and case sensitivity. More specifically, stick to "a-zA-Z0- 9_" characters. Use periods/full stops for file type only (i.e. `.csv`).
- Use delimiters to separate and make important metadata information (for example parameter values used in an analysis) retrievable further down the line. Use delimiters consistently, i.e. "_" to separate metadata to be extracted as strings later on and "-" instead of spaces or vice versa but do not mix. This makes names easy to match and search programmatically and easy to analyse.

Human readable

- Ensure file names also include informative description of file contents.
- Adapt the concept of the slug to link outputs with the scripts in which they are generated.

Easy to order by default

- Starting file names with a number helps.
- For data, this might be a date allowing chronological ordering.
- Make sure to use ISO 8601 format (YYYY-MM-DD) to avoid confusion between differing local dating conventions.
- For scripts, you could use a number indicating the position of the scripts in the analysis sequence e.g. 01_download-data.R

Organising projects for reproducibility

- Make sure you left-pad single digit numbers with a zero or you will end up with this:

```
10_final-figs-for-publication.R
1_data-cleaning.R
2_fit-model.R
```

Going back to our example project folder, in addition to the structure, note how good file naming and the use of slugs helps link scripts with their inputs and outputs.
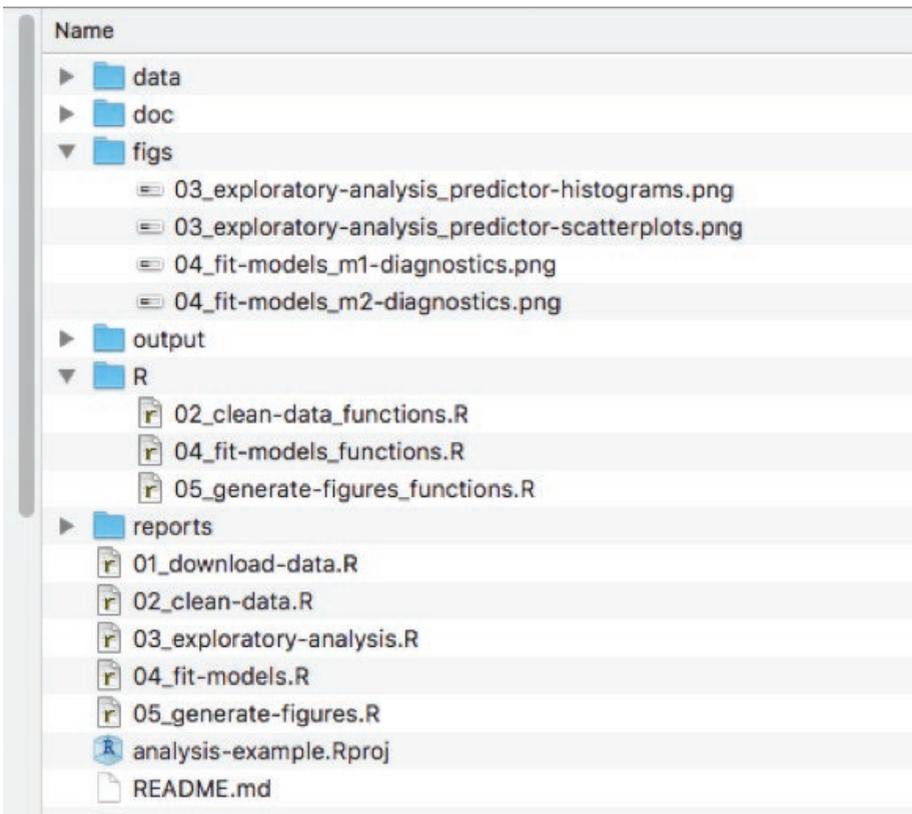
```
Name
  ▶  📁 data
  ▶  📁 doc
  ▼  📁 figs
          🗐 03_exploratory-analysis_predictor-histograms.png
          🗐 03_exploratory-analysis_predictor-scatterplots.png
          🗐 04_fit-models_m1-diagnostics.png
          🗐 04_fit-models_m2-diagnostics.png
  ▶  📁 output
  ▼  📁 R
          ® 02_clean-data_functions.R
          ® 04_fit-models_functions.R
          ® 05_generate-figures_functions.R
  ▶  📁 reports
      ® 01_download-data.R
      ® 02_clean-data.R
      ® 03_exploratory-analysis.R
      ® 04_fit-models.R
      ® 05_generate-figures.R
      Ⓡ analysis-example.Rproj
      🗋 README.md
```

Fig 2. Linking analysis associated files (e.g. R scripts with functions) and outputs (generated figures) through the use of consistent naming.

Now that you have set up a workflow and organised your project, you can go on to writing code!

Programming

Laura Graham, University of Southampton, UK

Writing code

Writing clear, reproducible code has (at least) three main benefits. Firstly, it makes returning to the code much easier a few months down the line; whether revisiting an old project, or making revisions following peer review. Secondly, the results of your analysis are more easily scrutinised by the readers of your paper, meaning it is easier to show their validity. Thirdly, having clean and reproducible code available can encourage greater uptake of new methods that you have developed.

To write clear and reproducible code, we recommend the following workflow:

write functions ➡ program defensively ➡ comment ➡ test ➡ document

In the following section we explain each part of this workflow along with some other tips for writing code. Although this workflow would be the 'gold standard', just picking up some of the elements can help to make your code more effective and readable.

> *For longer tasks plan your code using diagrams or pseudocode first.*
> – Samantha Price, Clemson University

Style guides

The foundation of writing readable code is to choose a logical and readable coding style, and to stick to it. Some key elements to consider when developing a coding style are:

- Using meaningful file names (see **Organising projects for reproducibility**), and numbering these if they are in a sequence.

GOOD

```
01-download-data.R
```

BAD

```
abc.r
```

- Concise and descriptive object names. Variable names should usually be nouns and function names verbs. Using names of existing variables or functions should be avoided.

Programming

GOOD
```
species_dat
```
BAD
```
data_from_site_one
sd
```

- Spacing should be used to improve visual effect: use spaces around operators (=, +, -, <-, etc.), and after commas (much like in a sentence).

- Indentation should be with spaces, not tabs, and definitely not a mixture of tabs and spaces.

The most important role of a style guide, however, is to introduce consistency to scripts. For more detailed information on style guides, and formalised approaches, see Google[2] and Hadley Wickham's R Style Guides[3], and the Python Style Guide[4].

Commenting code

How often have you revisited an old script six months down the line and not been able to figure out what you had been doing? Or have taken on a script from a collaborator and not been able to understand what their code is doing and why? An easy win for making code more readable and reproducible is the liberal, and effective, use of comments. A comment is a line of code that is visible, but does not get run with the rest of the script. In R and Python this is signified by beginning the line with a #.

One good principle to adhere to is to comment the 'why' rather than the 'what'. The code itself tells the reader what is being done, it is far more important to document the reasoning behind a particular section of code or, if it is doing something non-standard or complicated, to take some time to describe that section of code.

It is also good practice to use comments to add an overview at the beginning of a script, and commented lines of '---' to break up the script:

e.g. in R

```
# Load data -------------------------------------------------
```

> *Use feedback from others who use your code (such as labmates) to make the code more readable. For example, if your labmate misunderstands a function, rewrite the comments for that function.* – April Wright, University of Iowa

Programming

Portable code

When working collaboratively, portability between machines is very important, i.e. will your code work on someone else's computer? Portability is also important if code is being run on another server, for example a High Performance Cluster. One way to improve portability of code is to avoid using absolute paths and use only relative paths [5], ensuring that the script is run from the project root folder (see **Organising projects for reproducibility**). An absolute path is one that gives the full address to a folder or file. A relative path gives the location of the file from the current working directory. For example:

```
# Absolute path ----------------------------
"C:/project_root_folder/data/species_dat.csv"

# Relative path ------
"data/species_dat.csv"
```

Relative paths are particularly useful when transferring between computers because while I may have stored my project folder in 'D:/projects', you may have yours stored in 'C:/Users/My Documents'. Using relative paths and running from the project folder will ensure that file-not-found errors are avoided.

When working in R, RStudio projects can help aid portability of code. RStudio projects provide a self-contained coding environment to work in, and when opened, a .Rproj file sets the project working directory and saves the history and state of the project. RStudio projects can be created directly from the RStudio GUI and can be created in an existing folder, a new empty folder, or checked out from a version control repository (see **Version control**).

Writing functions

Often when you are analysing data, you need to repeat the same task many times. For example, you might have several files that all need loading and cleaning in the same way, or you might need to perform the same analysis for multiple species or parameters. The first instinct when faced with such a task would be to copy and paste

[2]https://google.github.io/styleguide/Rguide.xml accessed November 2017

[3]http://adv-r.had.co.nz/Style.html accessed November 2017

[4]https://www.python.org/dev/peps/pep-0008/ accessed November 2017

[5]https://en.wikipedia.org/wiki/Path_(computing)#Absolute_and_relative_paths accessed November 2017

Programming

previous blocks of code to repeat these tasks. This, however, is likely to introduce errors into your code, and therefore limit reproducibility. You can use functions to repeat tasks in a more standardised way. Using functions allows you to break your code down into modules, and therefore has the added advantage that it helps to provide better structure to scripts.

A function is a self-contained block of code that performs a single action. A function takes in a set of arguments, applies the action, and returns an object of any data type. A function should not rely on data from outside of the function, and should not manipulate data outside of the function.

Below are examples of how to define your own functions in R and in Python:

```
# Create a user-defined function in R:
square_number <- function(base){
        square <- base*base
        return(square)
}
# Calling a user-defined function in R
square_number(5)
```

Note that the function above takes in one number as an argument, `base`, squares this number, then returns one number as the output. The body of the function is enclosed within rounded brackets. You are asking R to return the result using `return(square)` (this is not strictly needed in this example, but using return saves problems when writing more complicated functions). In R, most functions are vectorised. This means that if you pass a vector to a function, the action will be applied to each element of the vector and the resulting new vector output.

```
# Create a user-defined function in Python:
def square_number(base):
    square = base*base
    return(square)

# Calling a user-defined function in Python:
x_square = square_number(5)
```

Programming

Again, the function takes in one number as the argument and outputs the square of this number. In Python, a colon defines the start of the function, and the body of the function is always indented (indentation marks blocks of code in Python). Python functions are not vectorised, so to get the output of a function applied to a list of numbers, you need to use the in-built map function:

```
#Calling a user-defined function element-wise on a list in Python:
squared_list = map(square_number, [1,2,3,4,5])
```

You can extend functions to include additional arguments. For example, if you wanted to define your function so that the output is the input number to a specific power, not just the square, you could include an additional argument to specify the exponent.

```
# Additional argument to R function:
exp_number <- function(base, power) {
          exp <- base^power
          return(exp)
}
```

You can expand upon these principles and include more arguments and apply more complicated functions to these arguments. The key to writing functions is to perform a single action per function, not rely on objects from outside the function, and not change objects outside the function. For more information on functions, see Hadley Wickham's Advanced R guide to functional programming.[6]

Defensive programming

Defensive programming is a technique to ensure that code fails with well-defined errors, i.e. where you know it shouldn't work. The key here is to 'fail fast' and ensure that the code throws an error as soon as something unexpected happens. This creates a little more work for the programmer, but makes debugging code a lot easier at a later date.

You can employ defensive programming on the `exp_number` function defined above. The function requires that both arguments are numeric, if you were to provide a string (e.g. a word) as input, you would get an error:

---

[6] http://adv-r.had.co.nz/Functional-programming.html accessed November 2017

Programming

```
exp_number("hello", 2)
Error in base^power : non-numeric argument to binary operator
```

If you add in a line of code to test the data type of the inputs, you get a more meaningful error.

```
exp_number <- function(base, power) {
   if(class(base) != "numeric" | class(power) != "numeric"){
      stop("Both base and power inputs must be numeric")
   }
   exp <- base^power
   return(exp)
}
```

```
exp_number("hello", 2)
Error in exp_number("hello", 2) :
  Both base and power inputs must be numeric
```

Although in this case, debugging the error would not have taken long anyway, in more complicated functions you are likely to either have less meaningful error messages, or code that runs for some time before it fails. By applying defensive programming and adding in these checks to the code, you can find unexpected behaviour sooner and with more meaningful error messages.

Testing scientific code

In the experimental sciences, rigorous testing is applied to ensure that results are accurate, reproducible and reliable. Testing will show that the experimental setup is doing what it is meant to do and will quantify any systematic biases. Results of experiments will not be trusted without such tests; why should your code be any different? Testing scientific code allows you to be sure that it is working as intended and to understand and quantify any limitations of the code. Using tests can also help to speed up the code development process by finding errors early on.

Most people will use informal testing by loading a function they have written and running ad hoc tests in the command line. Unit testing allows you to create formalised and automated tests of each section of your code. While unit testing may not be entirely necessary for many beginner coders, once you start writing your own functions, the recommended next step should be to start writing unit tests to go

Programming

along with these, and if you are writing packages these tests are essential. A guide to testing using the test-that package in R can be found in Hadley Wickham's online R Packages book[7]. For those using Python, the nose[8] package provides unit testing functionality.

Literate programming

The techniques described above are the building blocks for literate programming. Literate programming is a way of writing code so that it is clearly documented. Rather than writing code aimed at telling the computer what to do, you write code that tells other humans what you are instructing the computer to do (and why). Now that you have the building blocks, you can tie them together to achieve truly literate code. There are several tools that provide ways of doing this. For R, there is RMarkdown (see **Reproducible reports**), which allows you to integrate chunks of code into plain text written in the simple Markdown language. For Python, there are Jupyter[9] notebooks, which work in a similar way, integrating sections of code with sections of Markdown text. Markdown is a lightweight and straightforward markup language, which can be converted to many formats including HTML, PDF or even Word documents. It allows users to write plain text while using a few tags to format the text. The learning curve for Markdown is reasonably gentle. Here is some example Markdown text:

```
# This is header one
## This is header two

* here
* are
* some
* bulletpoints

*this text is in italics*
**this text is bold**
```

Markdown is in plain text (as opposed to rich text) format so it is easily transferred between platforms and integrated into a version control system (see **Version control**). There will be more about using these tools to write reproducible reports in the next section.

---

[7] http://r-pkgs.had.co.nz/tests.html accessed November 2017
[8] http://pythontesting.net/framework/nose/nose-introduction/ accessed November 2017
[9] https://jupyter.org/ accessed November 2017

Programming

Documenting and managing dependencies

François Michonneau, University of Florida, USA

Reproducibility is also about making sure someone else can re-use your code to obtain the same results as you.

For someone else to be able to reproduce the results included in your report, you need to provide more than the code and the data. You also need to document the exact versions of all the packages, libraries, and software you used, and potentially your operating system as well as your hardware.

R itself is very stable, and the core team of developers takes backward compatibility (old code works with recent versions of R) very seriously. However, default values in some functions change, and new functions get introduced regularly. If you wrote your code on a recent version of R and give it to someone who has not upgraded recently, they may not be able to run your code. Code written for one version of a package may produce very different results with a more recent version.

Documenting and managing the dependencies of your project correctly can be complicated. However, even simple documentation that helps others understand the setup you used can have a big impact. The following are three levels of complexity to document the dependencies for your projects.

Show the packages you used

With R, the simplest (but a useful and important) approach to document your dependencies is to report the output of `sessionInfo()` (or `devtools::session_info()`). Among other information, this will show all the packages and their versions that are loaded in the session you used to run your analysis. If someone wants to recreate your analysis, they will know which packages they will need to install.

Use packages that help recreate your setup

The checkpoint package in R provides a way to download all the packages at a given date from CRAN (The Comprehensive R Archive Network, cran.r-project.org). Thus, from the output provided by `sessionInfo()`, they could recreate your setup. It, however, makes two important assumptions: all your packages were up-to-date with CRAN at the time of your analysis; and you were not using packages that are not available from CRAN (e.g. the development version of a package directly from a git repository).

Programming

Another approach is to use the packrat package. This package creates a library (a collection of packages) directly within your analysis directory. It increases the size of your project as all the source code for the packages is included, but it ensures that someone can recreate more reliably the same environment as the one you used for your analysis. It also makes it easier because the installation of these packages is fully automated for the person wanting to have the same setup.

Use containers to share your setup

A step further in complexity is to use Docker (docker.com). With Docker you recreate an entire operating system with all the software, data, and packages needed for your analysis. It is more technical to set up but it allows you to distribute the exact same environment as the one you used. If you want others to be able reproduce your results, and your analysis depends on software that can be difficult to install, it is an option that might be worth exploring.

Reproducible reports

François Michonneau, University of Florida, USA

What is a reproducible report?

For the purposes of this guide, a report is a scientific document that contains not only the text that makes up the manuscript, but also the code that generates the figures and the statistics that are reported in your manuscript. Ideally, the report should be part of a self-contained project that may contain your data, your initial exploratory analyses, the final product, and the code needed to generate them.

In this context, a manuscript can be a scientific article, a conference presentation, a technical report, or a document to share your progress with your collaborators. The end product may not show any code and therefore it may not look like it was generated in a different way to any other documents.

Typically a report contains code for data manipulation, data analysis, and figure generation alongside the text that constitutes the heart of the report. If left unchecked, this mix can lead to a big mess that can be difficult to maintain and debug. Below is some advice on how to keep your report manageable.

Why a reproducible report?

By automating how the figures and the statistics in your report are generated, you are leaving a code trail that you, your collaborators, or your readers can follow, and that leads to your original data. This path to the raw data increases the transparency of your science. However, for the six-months-in-the-future you, your collaborators, and your readers, to be able to follow this path, it is important that you organise your code and your data files consistently (see **Organising projects for reproducibility**).

Not only does writing a reproducible report increase the transparency of your science, it reduces the mistakes that result from copying and pasting across software, helps keep results and models in sync, and allows you to provide interested readers with more information about the different approaches and analyses you tried before coming up with the final results.

These can be included as supplementary material or tagged in the history of your version control system (see **Version control**).

Reproducible reports

Writing reports using RMarkdown

Programming languages typically used by scientists for data analysis have libraries or packages that can be used to generate reproducible reports. The most popular ones are Jupyter Notebooks for scientists who primarily use Python and RMarkdown for those who use R. While they both share many commonalities, their implementation and everyday applications differ. Here, we focus on RMarkdown.

RMarkdown is a file format (typically saved with the Rmd extension) that can contain: a YAML header (see **Version control**), text, code chunks, and inline code. The rmarkdown package converts this file into a report, most commonly into HTML or PDF.

The **rmarkdown** package automates a multi-step process (Fig 3). Under the hood, it calls the **knitr** package that converts the Rmd file into a markdown (.md) file. In the process, **knitr** takes all the code chunks and the inline code, runs them through R (or other programs), captures their output, and incorporates them in the report. Afterwards, **rmarkdown** calls the pandoc program (an external program that is not related to R) that can take the markdown file and converts it into a variety of formats. For pandoc to generate PDF files, you will need a functional installation of LaTeX that you will need to install separately.

The **bookdown** package helps with numbering the figures and tables, and dealing with citations. As its name suggests, this package can be used to author books, but it is also well-suited to generating reports.
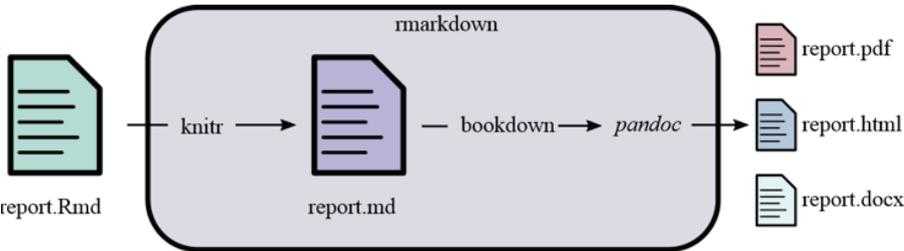

Fig 3. Relationships among the different R packages and tools used to generate reports from RMarkdown files.

Reproducible reports

Getting started

To start writing a report in **rmarkdown**, you can either open a file in your text editor of choice and save it with the .Rmd extension, or if you use RStudio, go to the File menu and then open a new RMarkdown file. You can then add a YAML header and various chunks as needed (see below).

The YAML header

The YAML header is at the top of your file; it is delineated by three dashes (**---**) at the top and at the bottom of it. They look like this:

```
---
title: "My reproducible report"
author: "BES"
date: "Today"
output: html_document
---
```

The YAML header is optional, but can be used to specify:

- the characteristics of your document: the title, authors, date of creation.
- the arguments to pass to pandoc to control the format of the output and additional information such as the bibliography file and the formatting of the list of references (see below).
- parameters for your report: for instance, you can specify a parameter such that your report will only use a subset of your data so that the final product will be generated quickly when you are developing the code for your project. Once your code is working, you can switch to the full dataset.

Code chunks

Code chunks are interspersed within the text of the report. They are delineated by three backticks (**```**) at the top and at the bottom. The top backticks are followed by a curly bracket that specifies: (1) the language in which the code chunk is written, (2) the name of the chunk (optional, but good practice), (3) **knitr** options that control whether and how the code, the output, or the figure are interpreted and displayed. Everything that comes after the name of the chunk has to be a valid R expression: the strings need be quoted, the arguments need to be separated by commas, and logical values (**TRUE/FALSE**) need to be capitalised.

Reproducible reports

Figures

The `knitr` package provides many options to finely control how your figures are going to be generated. Some of `knitr`'s options can be set individually for each chunk or be set globally. For a reproducible report, it is common practice to have a chunk at the beginning of the report that sets default options for the figures. It is also usually a good place to load all the packages you will need for your analysis. For instance the following chunk will ensure that:

- all the figures generated by the report will be placed in the `figures/sub-directory`

- all the figures will be 6.5 x 4 inches and centered in the text.

````
```{r figure-setup, echo=FALSE, include=FALSE}
knitr::opts_chunk$set(fig.path="figures/", fig.width=6.5,
                      fig.height=4, fig.align="center")
```
````

Additionally, this chunk will be named `figure-setup`, and we use the `echo=FALSE` option so the code for the chunk will not be displayed in the report, and use the `include=FALSE` option so no output produced by this chunk will be included in the report.

For our figures, we can now do

````
```{r sepal-width-length, fig.cap='Relation between sepal width
and length in three species of _Iris_.'}
library(tidyverse)
iris %>%
    ggplot(aes(x = Sepal.Width, y = Sepal.Length, color =
Species)) +
    geom_point()
```
````

When this file will be processed, it will create an image file (`figures/sepal-width-length.png`) with the default dimension and the caption specified by the value of the `fig.cap` argument. You can use markdown formatting (see **Programming**) within the captions of your figures. This figure will have the label `fig:sepal-width-length` that you will be able to use for cross referencing.

Reproducible reports

If you wish to incorporate a figure that is not generated by code (e.g. a photo of your field site or study organism), using the function `knitr::include_graphics()` takes care of many details for you, and generates labels and captions as if they were generated by code.

```
```{r iris-picture}
knitr::include_graphics("figures/iris.jpg")
```
```

Tables

To generate tables, `knitr` comes with the function `kable` that might be sufficient to make simple tables to represent data frames within your report. However, there are many packages that provide more sophisticated approaches to display and format tabular data within your reports. For an overview of the capabilities of different packages visit: hughjonesd.github.io/huxtable/design-principles.html

```
```{r iris-table}
iris %>%
    group_by(Species) %>%
    summarize(sepal_length = mean(Sepal.Length),
              sepal_width = mean(Sepal.Width)) %>%
     knitr::kable(caption = "Mean sepal width and length for
three species of _Iris_.")
```
```

Similarly to figures, when this is processed by knitr, the table will have the `tab:iris-table` label that can be used for cross-referencing.

Cross-references

If you want to refer to a certain figure or table in your report, you can use the `\@ref` (label) syntax. For instance

```
On average _setosa_ has wider and shorter sepals than the
other species(Fig. \@ref(fig:sepal-width-length), Table \@
ref(tab:iris-table)).
```

This will automatically provide the correct figure and table numbers for the sepal-width-length figure and the iris-table table.

Reproducible reports

Citations

You need two things: a BibTeX file that contains all the citations you use in your manuscript; and a CSL (Citation Style Language) file that specifies the format of your citations. You need to specify the names (and locations) of these files in your YAML header using

```
bibliography: reports/myrefs.bib and csl: reports/myrefstyle.csl.
```

Software citation managers such as Zotero or Mendeley provide options to generate BibTeX files for your citations. CSL files exist for most journals, and can be downloaded from: zotero.org/styles. This is a convenient search interface provided by Zotero, but you do not need to use Zotero to download or use these files.

Version control

Tamora James, University of Sheffield, UK

Scripts for reproducible code evolve over time, so keeping track of the changes that you make is important. You might need to access a specific version of your code to verify or reproduce the analysis used for a published paper, or to identify where you introduced a change that has caused problems in your code at a later stage.

Anyone who has wrestled with multiple versions of a document or script named by appending the word "final" will know how quickly such naming conventions can escalate into absurdity. Version control provides a structured and transparent means of tracking changes to code and other files. It was designed for use in software development and it is equally applicable to scientific programming. By recording snapshots of a project at successive points in time, you can create a record of your project's development while keeping your workspace clean. Version control also facilitates collaboration when used within project teams or when contributing to open source software projects.

Version control with cloud file storage services

Some cloud services such as Google Drive and Dropbox offer access to a file's version history, backing up file versions for a limited time, usually 30 days, and allowing restoration of previous versions within this time frame. While this can be a useful option for managing recent changes, for example to recover a previous version of a document, it is not a recommended route because there is limited control over revisions.

Version control software

Version control software is designed to help you manage your file revisions. Version control software runs directly on your computer, allowing you to manage files within your local file system. You can also use version control software to interact with external copies of versioned files if you choose.

Version control software is suitable for managing files that are largely text-based, such as code or text files with simple formatting like Markdown or LaTeX. It can also be used to store versions of other files such as graphics. Tracking changes in documents with extended file formats such as Word documents is better done using their built-in revision control, although you can use version control software to store snapshots of these files.

Version control

> Git Large File Storage (Git LFS), git-lfs.github.com, is an extension to git that can be used to manage large files by storing them outside your git repository, ensuring that the repository does not become unmanageably large.

Version control software is not recommended for use with very large files such as data files or databases because these generate excessive storage and download requirements. It is better to use other storage solutions for this type of file.

A wide range of version control software, both open source and proprietary, is available. This guide focuses on the widely used open source software called *git*.

Git is a good choice because it offers flexibility in how you can use it for version control. You can use it as a stand-alone tool to manage files on your own computer and, optionally, you can use it to connect to an external service to archive your files. Git can be used directly or through other software such as RStudio, so it is easily integrated into your normal workflow.

Git is a distributed version control system, which means that each user interacts with a stand-alone copy of the versioned files. This stand-alone copy is called a *repository* and it is usually a folder on your computer that contains all the work for a particular paper or project. Individual repositories are synchronised with each other by exchanging information about what changes have been made.

Version control repository hosting services

Git is sometimes confused with *GitHub*, which is a web-based hosting service for git repositories. There are a range of version control repository hosting services, including GitHub, GitLab, Bitbucket, and Savannah. These services allow you to create a remote copy of your local version-controlled project and to use it as an off-site backup and archive. It is easy to make your work available to collaborators through these services.

Version control repository hosting services offer a choice between *public* and *private* repositories. The former can be used for publicly accessible work, while the latter may be more suitable for sensitive or unpublished work. Public repositories are visible to anyone through the hosting service website. However, the owner of the repository retains control over the contents of the repository.

Version control

Currently, GitHub provides unlimited private repositories for educational users:
https://education.github.com/discount_requests/new
Your account needs to be associated with an academic/educational
email address.

Repository hosting services such as GitHub provide many other features including file preview and rendering, submitting change requests through the web browser, issue tracking, wiki-style documentation and website hosting. Some services offer desktop client software, which can be used to access your remote repository. GitHub is also integrated with the open access data repository service, Zenodo.

Getting started with git
- To get started you will need to download git: https://git-scm.com/downloads
- Once installed, you should configure git with your name and email address so that this information can be added to your version history. You will need to use the command line to do this, but you only need two simple lines of code:

```
git config --global user.name "Your Name"
git config --global user.email your.email@domain.com
```
- If you want to use RStudio's git integration, go to Options/Preferences › Git/SVN and make sure that the path to the git executable is correctly specified.

Git and the command line
While git can be used through software with a graphical user interface such as RStudio, there are occasions when you will need to use the command line. These include configuration and setting up remote repositories. You can access the command line (Terminal in MacOS and Linux or Command Prompt in Windows) from RStudio by going to Tools › Shell...

Git and GitHub will not only solve your version control needs, but will also streamline your project management and collaboration with colleagues.
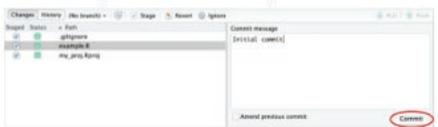– Francisco Rodríguez-Sánchez, Estación Biológica de Doñana (CSIC)

Version control

Fig 4. Steps in a typical version control workflow, and how to carry out these steps using git at the command line or through RStudio.

Version control

Version control workflow

In this section, you will get an overview of a basic version control workflow, get introduced to some of the terminology used and see some examples of how to carry out simple workflow tasks in git. You can find more help in the materials from the BES Best Practice for Code Archiving workshop[10] or by consulting some of the resources recommended at the end of this section. Don't worry if this seems complicated at first, it can take a bit of time to get comfortable with git!

A typical version control workflow (Fig 5) includes the following steps:

1. Set up a local version control repository

2. Write some code

3. Save your script to the repository

4. Choose files to be included in the version history

5. Create a record of your changes in the version history with a suitable message e.g. "Add new script to project"

6. Synchronise your changes with a remote repository
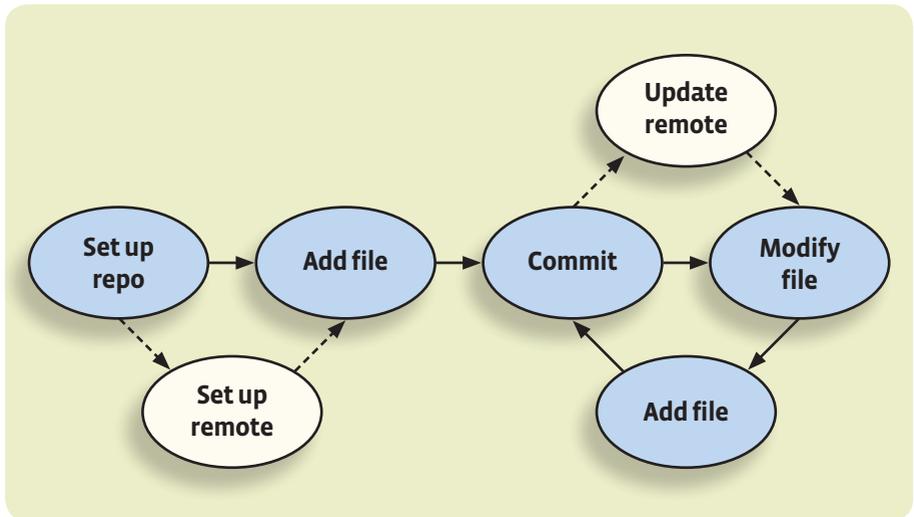
7. Repeat steps 2–7

Fig 5. A typical version control workflow

Version control

Using git to carry out these steps:

Setting up a repository - when you use git you interact with a set of versioned files called a *repository*. A git repository is simply a directory/folder on your computer with some hidden files for bookkeeping. You can create a git repository from scratch when you create an RStudio project. You can also clone git repositories from existing repositories (such as those on GitHub) or create them directly using the command line. If you have set up a version-controlled project in RStudio, the operations described below are available from the "Git" tab within the Environment/History panel (Fig 4).

Adding or staging files - git will only keep track of modifications to files that it has been asked to manage, so you need to tell git which files to include in your version history. You can do this by *adding* or *staging* files in your project directory.

Creating a commit - a *commit* is a snapshot of the changes to the repository. It forms part of the history of your project through time. When you create a commit using git you will be prompted to enter a *commit message*. This can be used to provide a description of the changes that you have made. Each commit is assigned a unique identifier that is stored together with the date, author, and commit message. This identifier can be used to specify a commit later.

Connecting to a remote repository - a local git repository allows you to track changes to files, storing the details in your version history and allowing you to go back to previous file versions, if necessary. If you want to create an external copy of your version history and share code with others, you should consider linking your repository to a *remote* repository. Synchronising your local repository to the remote repository will archive your code and provide a centralised store for your project, making it easy for you to share your work or to access it from another computer.

You can set up a remote repository on a hosting service such as GitHub or host it on your own server. If you are using a repository hosting service, make sure that you choose a private or public repository according to your needs. Please note that a public repository does not mean anyone can directly make changes to its contents; it simply means that the repository is publicly viewable. Private repositories are accessible only to those who have been granted access.

Pushing to the remote repository - this step sends the details of any commits which have been made locally to the specified remote repository, which can act as a backup of your work or a resource to share with others.

Version control

Making use of version control

Once you have a git version control workflow in place it can be useful in several different ways:

Viewing your version history - the version history that you build up as you commit your work forms a record of the changes that you have made to your code or documents. This can help you to track a project's development, understand reasons for past coding decisions, highlight major revisions and identify where bugs were introduced.

Returning to a previous commit - the real power of git will become clear when you need to return to a previous point in your revision history, for example to recreate figures for a paper or to run an analysis again. You can use the unique ID for a commit to tell git to return your working directory to the state captured by that commit. You can then work with your code as it was at that point in time.

Using version control to collaborate - there are a few alternative models for collaborating when using version control. If you use a distributed system such as git, each individual collaborator works with their own repository and you need to decide how to consolidate your work. A couple of possible workflows are:

1. Everyone connects their local repository to the same remote repository and must coordinate their changes. Changes to the central repository should be integrated into a user's local repository before they submit their own changes to the remote repository.

2. Individual contributors create a copy or *fork* of the main repository and use this as their remote repository. They must send a request to the maintainer of the original repository to incorporate their changes into the repository. This model allows a formalised review process before changes are integrated and is often used in open source projects.

Going further with version control

Git is a flexible and powerful toolkit that provides a wide range of advanced features such as branching, stashing and tagging. These topics are beyond the scope of this guide, but there are many online resources where you can read more about using these features in your workflow.

Version control

Git quick reference	
Add	the process of selecting a file for inclusion in version history
Branch	a separate set of changes to version history allowing users to work in parallel on the same files
Commit	a snapshot of changes to be added to version history
Commit message	user-specified description of the changes made in a commit
Conflict	a problem arising when changes from different sources cannot be combined automatically
Fork	a remote repository derived from another project that can be used for collaboration
Merge	combining changes that originate from different repositories or branches
Pull	an action that synchronises the local repository with remote changes
Push	an action that synchronises the remote repository with local changes
Remote	an external repository that can be synchronised with local changes
Repository	a directory containing the files under version control
Stage	the process of selecting a file for inclusion in version history

Archiving and citation of code and data

Mike Croucher, University of Sheffield, UK

In many areas of research, academic papers form only a part of scholarly output and they frequently do not contain enough information for reproducibility and detailed academic discourse. A more complete description of the research requires the publication of custom-written computer code, details on all software used (such as version information) and supporting data files.

Publication of academic code and data is becoming increasingly common, but there are currently no generally accepted standards for where and how these research objects should be published. Despite obvious shortcomings[11], it is common practice to upload code and data to personal websites that often disappear after a relatively short time. Even the more recent practice of linking to online version control repositories (e.g. GitHub, GitLab, Bitbucket or Savannah. See **Version control**) is insufficient, because there is no guarantee of permanence. This section proposes methods of code and data publication that are suitable for inclusion in the permanent scholarly record.

Six principles of software citation

When considering where to publish code and data, there are six principles of software citation:[12]

1. Importance - software should be considered a legitimate and citable product of research. Software citations should be accorded the same importance in the scholarly record as citations of other research products, such as publications and data; they should be included in the metadata of the citing work, for example in the reference list of a journal article, and should not be omitted or separated. Software should be cited on the same basis as any other research product such as a paper or a book, that is, authors should cite the appropriate set of software products just as they cite the appropriate set of papers.

2. Credit and attribution - software citations should facilitate giving scholarly credit and normative, legal attribution to all contributors to the software, recognising that a single style or mechanism of attribution may not be applicable to all software.

3. Unique identification - a software citation should include a method for identification that is machine actionable, globally unique, interoperable, and recognised by at least a community of the corresponding domain experts and, preferably, by general public researchers.

Archiving and citation of code and data

4. Persistence - unique identifiers and metadata describing the software and its disposition should persist even beyond the lifespan of the software they describe.

5. Accessibility - software citations should facilitate access to the software itself and to its associated metadata, documentation, data, and other materials necessary for both humans and machines to make informed use of the referenced software.

6. Specificity - software citations should facilitate identification of, and access to, the specific version of software that was used. Software identification should be as specific as necessary, using version numbers, revision numbers, or variants such as platforms where appropriate.

Licensing

Science is an inherently iterative process where we continuously build on work that came before. However, due to copyright-related legalities, others are not allowed to make any use of your work without a 'licence', even if you make it publicly accessible. A licence is an explicit statement that grants certain uses of a work. Therefore, it is critical to include a licence allowing people to reuse code when publishing it to an online repository. For text content (such as journal publications), one of the many Creative Commons licences is typically used. For computer code, one should always attach an 'open source' licence when publishing it. A guide on how to choose appropriate licenses for your work has been published by the Software Sustainability Institute.[13]

Attaching a licence to your work is straightforward. Once you have chosen the licence you wish to use, add a file called LICENSE or LICENSE.txt to your project that contains the license text. The website choosealicense.com, curated by GitHub, contains simplified information about available licenses along with the full licence text for you to copy and paste into your LICENSE or LICENSE.txt file.

Recommendations

To work towards satisfying these principles, it is advisable to use research data repositories such as Zenodo (zenodo.org), Open Science Framework (osf.io) or Figshare (figshare.com) when publishing code and data. Zenodo is a service that is free of charge for most usage, developed by CERN, and is available to researchers from all research fields. Figshare is a commercial option, owned by Digital Science, which offers a free usage tier. Software and data deposited with either service is provided with a globally unique Digital Object Identifier (DOI) and storage is guaranteed for

Archiving and citation of code and data

a long period of time. If used in association with GitHub, they automatically record information such as authorship and software version. Combined, these features help ensure that all the software citation principles can be met for many use cases.

For instructions on how to use Zenodo to archive code, see the materials from the BES Best Practice for Code Archiving workshop.[14]

Many universities also have institutional repositories for free code and data archiving. – April Wright, University of Iowa

---

[11] Martin Klein et al.: Scholarly Context Not Found: One in Five Articles Suffers from Reference Rot. See Resources
[12] Smith et al.: Software citation principles. See Resources
[13] Chue Hong & Tim Parkinson, Choosing an open-source licence. See Resources.
[14] https://github.com/BES2016Workshop/version-control accessed November 2017

Resources

Organising projects for reproducibility

Broman K. (2015) Organizing data in spreadsheets: http://kbroman.org/dataorg/

Data Carpentry Reproducible Research Committee (2016) File organization for reproducible research: http://www.datacarpentry.org/rr-organization1/

Kirchkamp O. (2016) Workflow of statistical data analysis: https://www.kirchkamp.de/oekonometrie/pdf/wf-screen2.pdf

McGill B. (2016) Ten commandments for good data management: https://dynamicecology.wordpress.com/2016/08/22/ten-commandments-for-good-data-management/

Noble W.S. (2009) A Quick Guide to Organizing Computational Biology Projects. *PLOS Computational Biology*, 5(7): e1000424. DOI: 10.1371/journal.pcbi.1000424

Wilson G. et al. (2017) Good enough practices in scientific computing. *PLoS Comput Biol* 13(6): e1005510. DOI: 10.1371/journal.pcbi.1005510

Programming

Loman N. & Watson M. (2013) So you want to be a computational biologist? *Nature Biotechnology* 31, 996–998, DOI: 10.1038/nbt.2740

Reproducibility in Science: http://ropensci.github.io/reproducibility-guide/

Sandve G.K. et al. (2013) Ten Simple Rules for Reproducible Computational Research. PLoS Comput Biol, 9(10): e1003285. DOI: 10.1371/journal.pcbi.1003285

Use of an R package to facilitate reproducible research: https://github.com/ropensci/rrrpkg

Reproducible reports

Hart E.M. et al. (2016) Ten Simple Rules for Digital Data Storage. *PLoS Comput Biol* 12(10): e1005097. DOI: 10.1371/journal.pcbi.1005097

Kitzes J., Turek D., Deniz F. (Eds.) (2018) *The Practice of Reproducible Research*. University of California Press, Berkeley: https://www.gitbook.com/book/bids/the-practice-of-reproducible-research/details

Knell, R. (2015) Introductory R Markdown: dynamic documents and reproducible research for beginners: http://www.introductoryr.co.uk/Reproducibility/Markdown_guide.html

Marwick, B. (2016). Computational reproducibility in archaeological research: Basic principles and a case study of their implementation. *Journal of Archaeological*

Resources

Method and Theory, 24(2) 424–4501. DOI: 10.1007/s10816-015-9272-9

Noble W.S. (2009) A Quick Guide to Organizing Computational Biology Projects. *PLoS Comput Biol* 5(7): e1000424. DOI: 10.1371/journal.pcbi.1000424

Sandve G.K. et al. (2013) Ten Simple Rules for Reproducible Computational Research. *PLoS Comput Biol*, 9(10): e1003285. DOI: 10.1371/journal.pcbi.1003285

The bookdown website: https://bookdown.org/yihui/bookdown/

The Reproducible Research CRAN Task View: https://cran.rproject.org/web/views/ReproducibleResearch.html

The RStudio Markdown website: http://rmarkdown.rstudio.com/

Wilson G. et al. (2014) Best Practices for Scientific Computing. *PLoS Biol* 12(1): e1001745. DOI: 10.1371/journal.pbio.1001745

Wilson G. et al. (2017) Good enough practices in scientific computing. *PLoS Comput Biol* 13(6): e1005510. DOI: 10.1371/journal.pcbi.1005510

Version control

British Ecological Society (2016) Guide to version control from BES Best Practice for Code Archiving workshop: https://github.com/BES2016Workshop/version-control

Dudler R., Git - the simple guide: http://rogerdudler.github.io/git-guide/

Chacon, S. & Straub, B. (2014) Pro git. Apress. https://git-scm.com/book/en/v2

Git reference/cheatsheets: https://git-scm.com/docs

Oh shit git! http://ohshitgit.com/

Software Carpentry Foundation (2016), Guide to Version Control with Git http://swcarpentry.github.io/git-novice/

Using Version Control with RStudio: https://support.rstudio.com/hc/en-us/articles/200532077-V

Archiving

Hong, N.C. & Parkinson, T. Choosing an open-source licence: https://www.software.ac.uk/choosing-open-source-licence

Klein M. et al. (2014) Scholarly Context Not Found: One in Five Articles Suffers from Reference Rot. *PLoS ONE* 9(12): e115253. DOI: 10.1371/journal.pone.0115253

Smith, A. M., Katz, D. S., Niemeyer, K.E., FORCE11 Software Citation Working Group (2016) Software Citation Principles. *PeerJ* DOI: 10.7717/peerj.2394

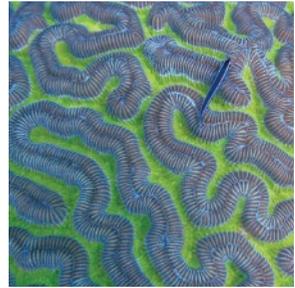# Acknowledgements

Image credits

Cover: David J. Bird

p9: Leejiah Dorward

p17: Leda Cal

p19: Isla Myers-Smith

p25: David J. Bird

p26: Leejiah Dorward

p37: Sonia Valladares

JOIN OUR COMMUNITY

Our vision is for a world inspired, informed and influenced by ecology. Membership is open to anyone, anywhere. Speak to us about how you can be involved.

hello@britishecologicalsociety.org
@BritishEcolSoc

MEMBERSHIP

- Member-only grants:

  Large Research grants – £20,000
  Small Research grants – £5,000
  Training and Travel – £1,000
  Outreach – £2,000
  Ecologists in Africa – £8,000

- Free access to ALL our journal content on the Wiley Online Library

- 25% discount on open access fees when publishing as first or corresponding author in our journals

- 10% discount when publishing in Ecology and Evolution

- Mentoring opportunities

- Preferential registration rates to all our events

- Ability to network with a global community of ecologists

- Personal development – join a committee or Special Interest Group, take part in a policy initiative or education consultation, propose a symposium or thematic topic

- The Niche, our quarterly print magazine sent to all members and for everyone interested in ecology

- Your monthly eBulletin packed with topical news, deadlines and opportunities to get involved

How to join

Membership only costs £22 per annum for students, retired members and those from low to lower-middle income countries. Prices for ordinary members start from £45 per annum.

Students:
UK Student Membership is free for one year, for anyone studying an ecology or ecology-related Undergraduate or Master's degree or if they are in the first year of their PhD. You must be registered to a UK bank account.

britishecologicalsociety.org/join-us

PUBLICATIONS

Journal of Ecology

journalofecology.org
@jecology

High-impact, broad reaching articles on all aspects of plant ecology (including algae), in both aquatic and terrestrial ecosystems.

Functional Ecology

functionalecology.org
@funecology

High-impact papers that enable a mechanistic understanding of ecological pattern and process from the organismic to the ecosystem scale.

Journal of Applied Ecology

journalofappliedecology.org
@jappliedecol

Novel, high-impact papers on the interface between ecological science and the management of biological resources.
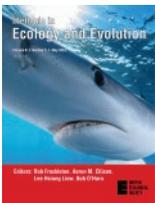
Journal of Animal Ecology

journalofanimalecology.org
@animalecology

Publishing the best animal ecology research that develops, tests and advances broad ecological principles.

Methods in Ecology and Evolution

methodsinecologyandevolution.org
@methodsecolevol

Promotes the development of new methods and facilitates their dissemination and uptake by the research community.

People and Nature

people-and-nature.org
@PaN_BES

A broad-scope open access journal publishing work from across research areas exploring relationships between humans and nature. **Now open for submissions!**
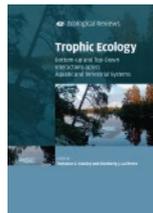
Guides to Better Science

bit.ly/GuidesToBetterScience

Promoting research excellence across a range of topics including peer review, data management, reproducible code and getting published. These free guides contain practical tips for researchers all over the world.

Ecological Reviews

bit.ly/EcologicalReviews

Books at the cutting edge of modern ecology, providing a forum for current topics that are likely to be of long-term importance to the progress of the field.