

Reproducibility of data analysis

Stephanie Palmer & Stefano Allesina & Graham Smith

September 12-14, 2017

Contents

Goals	1
Installation notes	1
Readability	2
Commenting your code	2
Writing readable code	3
DIY lottery	4
Save your seeds	5
Introduction to version control	6
Git	7
Everyday Git	8
Summary of Daily <code>git</code> Commands	10
Remote repositories	11
References and readings	14
Data Challenge	14

Goals

This tutorial will cover methods for making your code reproducible, both by you and by others. Producing results that are reproducible by others is the very essence of science, and writing code that is reproducible by **you** is just the first step. In particular, we will discuss methods to ensure that you can: reproduce your calculations precisely, make your code readable, track changes to your code, and make your code freely accessible to the wider world via [github](#). Along the way, you will get an introduction to stochastic processes and how they are used to model biological variability. By the end of this tutorial, you should know why it is important to save your seeds and merge your branches. You should also know why reproducible coding practices can help you *now*, even if (indeed, particularly if) you are just learning to code.

Installation notes

For this tutorial, relatively up-to-date versions of R and RStudio are needed. To install R, follow instructions at [cran.rstudio.com](#). Then install Rstudio following the instructions at [goo.gl/a42jYE](#). Download the `ggplot2`, `cowplot`, `stats`, and `RMKdiscrete` packages. You should also install the UNIX utility `git` for [OSX](#) or [Windows](#).

For the bonus data challenge, you will also need to install the Unix utility `libcurl`. If you haven't done so already, you can find installation instruction links here by first clicking on the libcurl [Download Wizard](#). You only need to install the pure binary libcurl package.

Readability

When you begin programming, you have written just a few programs and functions and it might be possible to hold all of your naming conventions and little tricks in your mind at once. However, wait just a few days – let alone a few months – and you’ll have a hard time deciphering, perhaps even running *your own* code if you don’t write readable programs. Best practices in modern computational science also dictate that you share your code whenever you publish a result. This means that other people need to be able to read your code; it should run on other computers with other folks sending the input data and specifying or *changing* parameters. This might seem daunting, but a few simple habits will aid in keeping all of your code parsable by other humans, including your own future self.

Commenting your code

The simplest way to make your code more readable is to write comments. Comments begin with `#` and continue to the end of the line. There are several different types of comments, and each explains what code does, how it’s done, or why you did it. For our purposes, we’ll divide comments into two broad categories: block comments and in-line comments. Block comments take up the whole of one or more lines and describe meaningful units comprising many lines of code. In-line comments occupy the end of a line of code, or are sandwiched between lines of code, and describe only one or few lines of code.

Block comments

Each meaningful part of your program should have a block comment explaining the purpose of that part (e.g. functions, scripts, the whole program).

```
# Modeling a Society of Foos
#
# Author:      Barry Allen
# Created:     2017 Sep 15
# Last Modified: 2017 Sep 16
#

make_a_foo <- function() {
  # Constructs a foo.
  #
  # Returns a foo object.

  return(foo)
}

crown_a_foo <- function(foo) {
  # Create a new royal foo from an ordinary foo.
  #
  # Args:
  #   foo - a foo object with no royal status
  # Returns a royal foo
  foo$is_royal = TRUE
  return(foo)
}
```

First, at the top of your file you should include a block comment that describes who wrote the code, when it was last updated, what it does, and how and why someone would use it.

Similarly, the first lines of any function should be comments. These comments should describe what the function does as well as its inputs and outputs. Ideally name and describe each of the function's arguments individually, as well as the return value. For advanced users, there are packages, e.g. `roxygen2`, that will automatically convert such comments into documentation in the `man` folder, that can then be accessed by typing `?` or `help()`.

If you are writing a script, you should include a comment at the top of every meaningful code block. These blocks are somewhat like paragraphs, and like paragraphs, there is no one right way to block your script. A good indicator of whether or not a block is a meaningful block is whether or not you can write a simple comment for that block.

Block comments should describe what you *intend* the code to do, not what it does in detail. The former can (and should!) be done even before you write the code (like unit tests discussed in defensive programming). These comments will not only be concise, they can actually help you write better code. Comments that describe how the code works in detail are simply repeating the actual code in words, and probably more clumsily. This is redundant; don't do it.

In-line comments

Complex lines of code should include in-line comments to help the reader understand what the line is doing.

```
z <- (x + y) / 2 # Average x and y
# Average of three averages
weird <- (((k + j) / 2) + ((x + y) / 2) + ((b + c) / 2))/3
```

Notice there is a limit to how much these in-line comments can help. They successfully translate the code into English (as opposed to simply repeating it), but absent any context, we can't understand what is happening, or why. That is what the block comments should provide.

Writing readable code

Even more readable than complex code with in-line comments is simple code that doesn't need comments. Whenever possible write Really Obvious Code (ROC). Tips for writing ROC:

- **Use good names** Use function and variable names that are self-explanatory. For example, `random_locations_of_N_spiders_in_a_box.R` is a much better function name than `EEK.R`. Don't worry about forgetting such a long name, or even typing it in. In RStudio, you can simply type `random[TAB]` to see a list of all functions you've defined whose name starts with `random`, so the extra characters don't waste time.

- **Do one thing at a time** This:

```
foo <- make_a_foo()
king <- crown_a_foo(foo)
dethrone_a_foo(king)
```

is much more readable than `dethrone_a_foo(crown_a_foo(make_a_foo()))` or even `make_and_crown_and_dethrone_a_foo()`. Each line is a sentence, and when you do many things on the same line, you run the risk of run-on sentences.

- **Don't use magic numbers** When programmers talk about magic numbers, they don't mean 7 (necessarily). They mean any number that's just sitting in your code, unnamed. This is similar to "use self-explanatory variable names." `area <- 5 * 3` is not as clear as `area <- width * height`.

DIY lottery

To practice writing readable code, let's write a short program to generate a draw from a lottery. The lottery is a somewhat peculiar one: Each ticket for the lottery is a buyer-determined string of 30 1's and 0's. The winning lottery number is drawn by flipping a coin 30 times and reporting a "1" for heads and a "0" for tails. The same bent coin is always used, and it has a probability of 0.1 of turning up heads and a probability of 0.9 of turning up tails. The prize money for the lottery is \$10,000,000, and each ticket costs only \$1.

Let's make sure to load the **stats** package.

```
library(stats)
```

Next, let's explore how we simulate flipping this bent coin once. We're going to use one of R's random number generator functions **runif**, which will give you uniformly distributed numbers on the interval $[0,1]$. Here's one way to do it:

```
# define the probability of heads
p_heads <- 0.1
# draw a random number from the interval [0,1]
random_val <- runif(1)
# threshold the random number based on p_heads to decide the sign of the flip
if (random_val < p_heads) {
  coinflip <- 1
} else {
  coinflip <- 0
}
# 1 is for heads, 0 is for tails
```

Can you see any ways to make this code even more Really Obvious?

Exercise

Write a function in R that will simulate one random draw for the winning lottery ticket. Let the inputs to the function be the number of digits in the lottery ticket (default value 10) and the probability of heads (default value 0.1). The function should output a sequence of 1's and 0's. Use all of the best practices for commenting your code.

Each flip of a coin like this with probability, p , of heads is an example of a Bernoulli trial, which is the technical term for an experiment with only two output states, success or failure. The number of heads in the sequence of independent coin flips generated by our lottery will follow a binomial distribution

$$P_n(k) = \binom{n}{k} p^k (1-p)^{n-k},$$

where p is the probability of heads (1's), n is the length of our lottery ticket, and k is the number of heads in the ticket. The prefactor $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is called the binomial coefficient and describes the number of unique ways of placing k identical objects in n bins. Capital " P_n " represents the probability distribution of k heads out of n tosses.

You can use the function **rbinom** to generate draws from a Bernoulli distribution, since a Bernoulli trial is just a binomial trial with only one coin flip. In our lottery, this code would amount to flipping the coin n_{flips} times.

```
n_flips <- 10
rbinom(n_flips, 1, p_heads)
```

```
# [1] 0 0 1 0 0 0 1 0 0 0
```

Exercise

If you could only buy one ticket for this lottery, which one would you buy? Modify your code (perhaps use `rbinom`) and run it several times (maybe 100 or even 1000) to sample from lots of possible lottery outcomes. Did you pick a good number? How many times did your number come up?

Save your seeds

Compare a draw from your lottery with your neighbor. Do you draw the same sequence of random lottery tickets? Why not? If you wanted to reproduce the *exact* same output from your lottery each time you decide to reset it, you'll need to know a little more about how R's random number generator (RNG) works. Try typing:

```
? RNG
```

That should open documentation in the “Help” pane. You will notice that the function `RNGkind` is the interface for querying the current state of the RNG. Let's find out what the current settings are:

```
RNGkind()
```

```
# [1] "Mersenne-Twister" "Inversion"
```

The first part is the RNG algorithm, the second specifies the algorithm for transforming uniformly distributed random numbers into random samples from the normal, or Gaussian, distribution. The twister algorithm based on Mersenne prime numbers, $M_n = 2^n - 1$, where n is also prime, is a state-of-the-art pseudo-random number generation scheme, developed by Matsumoto and Nishimura in 1997. NB: RNG's should technically be called *pseudo*-random number generators or PRNG's, in part because they all have some period after which they will produce exactly the same sequence. The trick is to find an algorithm with a period so long you'll never notice the “P” in the “PRNG”. The Mersenne Twister algorithm has a period of $2^{19937} - 1$ and passes many statistical tests for randomness.

The seed to an RNG is usually a large integer that provides an initialization the RNG algorithm. Scrolling down to “Note” in the “Help” pane, you'll learn that the seed to R's RNG is set by the current time and the process ID. That means that your simulation results will depend on when you start your R session, run your code, and even some local information in your processing environment. Compare the output of `runif` with your neighbor.

```
runif(5)
```

Starting an RNG with the same seed will produce exactly the same sequence of random numbers; an RNG spits out random numbers, but not noisy ones. To reproduce your simulation results precisely when you use an RNG, you'll want control of that seed. R uses `set.seed` which takes a small integer as input and generates, deterministically, all the random seeds necessary for your RNG algorithm.

```
set.seed(19937)
runif(5)
runif(5)
set.seed(19937)
runif(5)
```

Exercise

Test whether or not you and your neighbor get precisely the same sequence of numbers when you use the same seed. Will this work if you use different RNG algorithms? Try changing your RNG algorithm using `RNGkind` and compare your results with your neighbor, when you use the same seed. If you wanted to be able to instruct someone to reproduce your exact simulation results using R's RNG, what would you need to tell them?

You can use `RNGkind` to set or query both the RNG and normal algorithms. You can save this information along with the current value of `seed` using something like:

```
seed <- 19937
set.seed(seed)
seed_used <- seed
RNGkind_used <- RNGkind()
save("seed_used", "RNGkind_used", file=RNGinfo_for_mycode)
```

When you are ready to share your code with others, you should also save all the version information for your current R package and libraries to this same file.

Introduction to version control

Version control is a way to record and organize changes to a set of files and directories (e.g., the directory containing one of your projects). Over time, a version control system (VCS) builds a database storing a history of snapshots of your project (your *repository*), making the whole history of the project available.

When you start working on a new project, you tell your VCS to keep track of all the changes, additions, and deletions. At any point, you can *commit* the changes, effectively building a snapshot of the project in the repository. This snapshot of the project is then accessible—you can recover previously committed versions of files, including metadata such as who changed what, when and why. Also, you can easily start tracking files for an existing project.

Version control is especially important for collaborative projects: everybody can simultaneously work on the project, even on the same file. Conflicting changes are reported, and can be managed using side-by-side comparisons. The possibility of *branching* allows you to experiment with changes (e.g., shall we rewrite the introduction of this paper?), and then decide whether to *merge* them into the project.

Why use Version Control?

If you ever collaborate on writing code, then version control is for you. If you ever horribly break your project and need to move quickly back in time two days (or two weeks), then version control is for you. If you ever need to share your code publicly, then version control is for you. Most likely, version control is for you.

Version control is useful for small, and essential for large collaborative projects. It vastly improves the workflow, efficiency and reproducibility. Without it, it is quite easy to lose track of the status of a manuscript (who has the most recent version?), or lose time (I cannot proceed with my changes, because I have to wait for the edits of my collaborators).

Many scientists keep backup versions of the same project over time or append a date/initials to different versions of the same file (e.g. various drafts of the same manuscript). This manual approach quickly becomes unmanageable, with simply too many files and versions to keep track of in a timely and organized manner. Version control allows you to access all previously committed versions of the files and directories of your project. This means that it is quite easy to undo short-term changes: Bad day? Just go back to yesterday's version! You can also access previous stages of the project: "I need to access the manuscript's version and all the analysis files in exactly the state that they were in when I sent the draft for review three months ago." Checking out an entire project at a certain point in time is easy with a version control system but much more difficult with Dropbox or Google Drive.

Version control makes it trivial to host your code publicly (e.g. on Github or Bitbucket) and to share a robust link to your code in any publication.

Version control might look like overkill at first. However, with a little bit of practice you will automatically run a few commands before you start working on a project, and again once you are done working on it. A small price to pay, considering the advantages. Simply put, using version control makes you a more organized and efficient scientist.

Stefano's testimonial: "Our laboratory adopted version control for all our projects in 2010, and sometimes we wonder how we managed without it."

Git

For this introduction to version control, we use `git`, which is one of the most popular version control systems. `git` is also free software, and is available for all computer architectures. Many good tutorials are available online, and many websites will host your repositories for free.

Other options you might want to consider are `Mercurial` (very similar to `git`) and Subversion Version Control `svn`, which is an older system, but still widespread.

There are two main paradigms for VCSs allowing multiple users to collaborate: in a *centralized* VCS (e.g., `svn`), the whole history of a project is stored exclusively on a server, and users download the most current snapshot of the project, modify it, and send the changes to the server; in a *distributed* VCS (e.g., `git`), the complete history of the repository is stored on each user's computer.

`git` was initially developed by Linus Torvalds (the "Linu" in Linux), exactly for the development of the Linux kernel. It was first released in 2005 and has since become the most widely adopted version control system.

At first, we work with *local* repositories, meaning that all the files are stored exclusively on your computer. `git` by default works with local repositories. Later, we introduce *remote* repositories, which are also hosted on a web server, making it easy for you to share your projects with others (or work on the same project from different computers). These remote repositories can be hosted on such well known sites as Github and Bitbucket, and `git` will play nicely.

Configuring git

First, open a terminal window.

The first time you use `git`, (or whenever you install `git` on a new computer), you need to set up the environment. To store your preferred user name and email, type:

```
$ git config --global user.name "Charles Darwin"
$ git config --global user.email crdarwin@royalsociety.org
```

You can set up your preferred text editor, which will be used to write the messages associated with your commits. If you are familiar with text editors and already have a favorite (e.g. `emacs` or `vim`), feel free to use it. Otherwise, Windows users please use `notepad` and Mac users please use `nano`.

```
$ git config --global core.editor nano
```

Again, Windows users please replace `nano` with `notepad` in the above.

To check all of your settings and see all available options, type:

```
$ git config --list
```

How to get help in Git

For a brief overview of common `git` commands, open your terminal and type:

```
$ git help
```

Now try typing:

```
$ man git
```

you will see that the name of the program is `git` - the stupid content tracker. Git behaves stupidly in the sense that the system tracks any content without being selective (which can be a good and bad thing). The manual page contains a description of all the commands but it is much more pleasing to read them online.

Everyday Git

To illustrate the basic operations in `git`, we consider the case of a local repository: all the versions are stored only in your computer, and we assume that you are the only person working on the project. Once familiarized with the basics of `git`, we introduce the use of remote repositories for collaborative projects.

Workflow

Here is the typical day in the life of a `git` user:

When starting a new project, 1) create a directory using terminal and initialize a repository. 2) Start working on your file until you reach a milestone or until you are done for the day. 3) Check what has changed since your last snapshot. 4) Decide which files to add to the new snapshot of the project (selected files, or everything). 5) Create the snapshot by committing your changes, including a detailed description. 6) Start changing the files again, add them to the new snapshot, and commit.

As you can see, there are only a few commands that you need to master for everyday work. Let's try our hands at this workflow and create a simple repository. Open a terminal window and create a new directory called `git_test`.

```
$ mkdir git_test
$ cd /git_test
$ mkdir originspecies
$ cd originspecies
$ pwd
$ git init
```

where we have moved to a newly created directory “originspecies”, and run the command `git init`, which initializes an empty repository. You can set up a `git` repository for an existing project by changing to the directory containing the project and typing `git init` in the terminal. We recommend to always run `pwd` before initializing a `git` repository to confirm that you are in the correct directory. The last thing you want is to track changes to your entire computer because you happened to be in your root directory.

We can check the status of the repository by running:

```
$ git status
```

Now we create our first file:

```
$ touch origin.txt
```

and then start editing it. We could use a text editor, but for the moment let's stick to the command:

```
$ echo "An abstract of an Essay on the Origin of Species..." > origin.txt
```

and check that our command went through:

```
$ cat origin.txt
```

`git` does not track any file unless you tell it to do so. We can set the file `origin.txt` for tracking with the command:


```
$ git add origin.txt
```

We can see that the status of the repository has changed:

```
$ git status
```

Every time we want to signal `git` that a file needs to be tracked, we need to add it to the index of files in the repository, using the command `git add FILENAME`. If we want to simply add all the files in a directory and subdirectories, use `git add .` (where the “.” means “current directory”). To add all the files contained in the repository directory (including “hidden” files), use `git add --all`.

Note that `git` and similar systems are ideally suited to work with text files: `.csv` and `.txt` files, code (`.R`, `.py`, etc.), LaTeX manuscripts (`.tex`, `.bib`), etc. If you track *binary* files, `git` will simply save a new version any time you change it, but you will not be able to automatically see the differences between different versions of the files.

Once we are finished creating, deleting and modifying our files, we can create a snapshot of the project by *committing* the changes. When should one commit? `git`’s motto is “commit early, commit often”. Every time you commit your changes, these are permanently saved in the repository. Ideally, every commit should represent a meaningful step on the path to completing the project (examples: “drafted introduction”, “implemented simulation”, “rewritten hill-climber”, “added the references”, ...). As a rule of thumb, you should commit every time you can explain the meaning of your changes in a few lines.

Now it’s time to perform our first commit:

```
$ git commit -m "started the book"
```

where `-m` stands for “message”. If no message is entered, `git` will open your default text editor, so that you can write a longer message detailing your changes. These messages are very important! In fact, you can use them to navigate and understand the history of your project. Make sure you always spend a few extra seconds to document what you did by writing a meaningful and detailed message. In a few months, you will have forgotten all the details associated with a commit, so that your future self will be grateful for detailed messages specifying how and why your project has changed.

The history of the repository can be accessed by typing:

```
$ git log
```

The long line of numbers and letters after the word `commit` is the “checksum” associated with the commit (i.e., a number `git` uses to make sure that all changes have been successfully stored). You can think of this number as the “fingerprint” of the commit.

Now, let’s change a tracked file:

```
echo "On the Origin of Species, by Means of Natural Selection, " > origin.txt
echo "or the Preservation of Favoured Races in the Struggle for Life" >> origin.txt
```

a much more powerful, albeit a bit long, title. The repository has changed and we can investigate the changes using `status`:

```
$ git status
```

showing that a) a file that is being tracked has changed, and b) that these changes have not been *staged* (i.e., marked) to be committed yet. You can keep modifying the file, and once you are satisfied with it, use `git add` to make these changes part of the next commit. For example:

```
$ git add .
$ git commit -m "Changed the title as suggested by Murray"
```

You now see the new commit in the history of the repository:

```
$ git log
```

That's it! For 99% of your time working with `git`, all you need to do is to follow these simple steps:

Summary of Daily `git` Commands

Creating an entirely new project:

```
$ mkdir newproject
$ cd newproject
$ git init # initialize repository
```

gitifying an existing project:

```
$ cd existingproject
$ git init
```

Once you already have a version controlled project in `git`, your daily routine:

- 1) Work! Make new files, change files, etc.
- 2) Add new or changed files to the snapshot:

```
$ git add [FILENAME]
```

- 3) Check the local repo's status to see if you forgot to add any files

```
$ git status
```

- 4) Commit the snapshot

```
$ git commit -m "my descriptive message"
```

These are a few of the most basic commands, but `git` allows you to do much more. We explore a few of the more advanced features in the next sections.

Exercise

Create a local `git` repo. Create three empty files inside the repo called `spiders`, `snakes`, and `monkeys` (e.g. by `touch FILENAME`). Compare the output of `git status` before and after adding each of the files. Commit your changes. Don't forget to use a descriptive message! What does `git status` say after the commit?

Keep this repo around, we'll come back to it.

Moving and removing files

When you want to move or remove files or directories under version control, you should let `git` know, so that it can update the index of files to be tracked. Doing so is quite easy: simply put `git` in front of the command you would run to perform the operation:

```
$ git rm filetoem.txt
$ git rm *.txt
$ git mv myoldname.txt mynewname.csv
```

If you run `git status` and see you've accidentally added a file you didn't want to track (e.g. your data) you can use `git rm --cached [FILENAME]` to unstage a file (i.e. prevent it from being committed). NB: Even on your local machine, it's a good idea to store data *outside* your `git` repository (especially if it's sensitive medical data) so that you don't accidentally add it to your repository. Remember: `git` remembers everything.

Deleting changes and reverting to last commit

Sometimes, changes can go horribly wrong, such that you would like to scrap everything, and abandon all the changes you made to one (or more) file(s). This can be accomplished by typing:

```
$ git checkout filetoreset.txt
```

If you want to scrap *everything*, you can type `git log` to see a list of previous commits, each of which will have a [REALLY LONG SEQUENCE OF CHARACTERS] that uniquely identifies that commit snapshot. To scrap all your changes and go back to the state of your project at the time of that commit, just type

```
$ git checkout [REALLY LONG SEQUENCE OF CHARACTERS]
```

Once you run this command, the file(s) will be in the last version you committed. Use this command with caution, as the changes you made are irrevocably lost.

Exercise

Play around with the files you made in the last exercise. Add some names of spiders, snakes, and monkeys to their respective files. What is the difference between the output of `git status` now and in the previous exercise? Commit these changes. Now try renaming `snakes` to `serpents` using `git mv`. Try removing `spiders` using `git rm`. What does `git status` say? Does `spiders` actually disappear from the folder when you remove it from the repository? What happens when you commit your changes?

We don't want to be mean to the poor spiders, so revert to the commit where you had the three files `spiders`, `snakes`, and `monkeys`, and each had a list of names written inside them. (I hope you actually committed then, and I hope you gave that commit a descriptive name...)

Remote repositories

So far, we have been working with local repositories, hosted in only one computer. When collaborating (or when working on the same project from different computers), it is convenient to host the repository on a server, so that the collaborators can see your changes, and add their own.

There are two main options: a) setup your own server (this is beyond the scope of this introduction); b) host your repositories on a website offering this service (for a fee, or for free).

The most popular option for hosting open source projects, where the whole world can see what and when you commit, is [GitHub](#), which will store all your public repositories for free. In fact, as a student you can get private repositories for free! The [Student Developer Pack](#) comes with access to lots of other goodies, but private repositories make it a necessity. They're useful in the early stages of your project when you're paranoid about anyone, let alone the entire world, seeing your hasty hacks. But remember to publicize your repository when you publish! (Note: probably still not HIPAA compliant)

After the initial setup (which is specific to the service you use, and thus not treated here), you only need to add two new commands to your `git` workflow: `pull` and `push`. When you want to work on a project that is tracked by a remote repository, you `pull` the most recent version from the server and work on your local copy using the commands illustrated above. When you are done, you `push` your commits to the server so that other users can see them. Your workflow will look like:

- 1) Wake up in the morning, get some coffee (or tea!), settle in at your computer, and type:

```
$ git pull
```

This will grab any changes an enterprising, insomniac collaborator `pushed` at 4am this morning. Or it will grab the changes you `pushed` from your laptop at 4am this morning. (note the word `push`!)

- 2) Get to work! Everything works just like it does for a local repository (because you do have a local repository, in addition to the remote repository). When you create or modify files, either add them individually or run:

```
$ git add --all
```

- 3) When you've hit a milestone, or loosely when you can explain what you've done in a line or two, **commit** your changes

```
$ git commit -m "A meaningful message"
```

- 4) Repeat steps 2 and 3 until...

- 5) "Ok, everyone should see my changes." It's time to send your changes to the remote repository:

```
$ git push
```

How often you should **push** will vary by project. If you're the only one working on your branch (see below), there's no reason not to **push** after every **commit**. If your computer abruptly transcends this existence, you'll only lose the progress since your last **push**.

Note that the interaction with the server only happens when you **pull** and **push**. In between these commands, you are in control, and can decide when and what to communicate to your collaborators.

Exercise (for at home!)

Make a GitHub account! You can even create a GitHub repository and push your local repository to that new GitHub repository. Even though you created this remote repository after your local repository, are your previous commits saved?

Branching and merging

Most VCSs allow you to *branch* from the main development of the project (i.e., experiment freely with your project without messing up the original version). It is like saying "Save as" with a different name, but with the possibility of easily merging the two parallel versions of the file. Typical examples of a branching point in scientific projects are: a) you want to try a different angle for the Introduction of your manuscript, but you are not sure it's going to be better; b) you want to try to rewrite a piece of code, to see whether it will be faster, but surely you do not want to ruin the current version that is working just fine; c) you want to keep working on the figures, while your collaborator is editing the manuscript. In these cases, you are working on an "experimental feature", while leaving the main project unaltered (or while other people are working on it). Once you are satisfied with your changes, you would like to *merge* them with the main version of the project.

Frankly, you should always be working on a branch.

In **git**, creating a new branch is quite easy. Suppose that you are working on a project, and that you have never branched before. Suppose that you want to try amending a piece of the code (for example, to make it run faster), but don't want to touch the code that is already working. You need to *branch*:

```
$ git branch fastercode
```

creates the new branch **fastercode**. Now you can type

```
$ git branch
```

to see a list of all the branches in your repository, including the default **master** and the newly created **fastercode**. To switch to your newly created branch, type

```
$ git checkout fastercode
```

Now you can commit as usual, but the changes will not affect the **master** branch.

```
$ git commit -m "Managed to make code faster"
$ git log
[commit log including "Managed to make code faster"]
$ git checkout master
$ git log
[commit log NOT including "Managed to make code faster"]
$ git checkout fastercode
$ git log
[new commit still saved!]
```

To summarize, here's how you work with branches:

- Create a new branch

```
$ git branch mybranch
```

- See list of branches, including indication of which branch you're currently on.

```
$ git branch
```

- Move to new branch

```
$ git checkout mybranch
```

- Start working on the branch. Everything described in the previous sections still works, just affecting the current branch rather than the **master** branch. (Note: when **pushing** the new branch, **git** will prompt you to set the upstream branch. Since you created this branch locally, the remote has no corresponding branch, so the line **git** prompts you to write will create the branch remotely, and then push local branch into the remote branch. "upstream" just refers to the remote branch corresponding to the local branch).

- Once you are satisfied, and want to include the changes into the main project, move back to main trunk

```
$ git checkout master
```

- Merge the branch

```
$ git merge mybranch -m "message for merge"
```

- [optional] delete the branch

```
$ git -d mybranch
```

Exercise

It's time for war. Create (at least two) new branches called **monkeys_are_the_best** or similar, and appropriately wage war on your enemies and shore up defenses (more monkeys might help – Zizou, for example, is the best monkey name). Switch between branches to wage war on all sides. But don't be mean, then I'll feel bad for writing this exercise. Wage a *nice* war.

Once you're satisfied with the carnage (the *nice* carnage), switch back to the master branch and try your hand at merging the branches into master. The first one should be simple! The second one? Less so. If both are simple, go back and wage better war (i.e. modify the same line in two branches). How do you resolve a merge where there are conflicts? This is in general not easy, but **git** tries to help as much as possible.

References and readings

Journal articles

All of the data used in this tutorial come from original research papers that are in the `readings` folder. Also in the `readings` folder, you will find an article by Roger Peng arguing for setting standards for reproducible research in computational science. It's a short article that we hope you will read and adopt as best practices for your own work.

Books and tutorials

There are very many good books and tutorials on `git`. We are particularly fond of *Pro Git*, by Scott Chacon and Ben Straub. You can either buy a physical copy of the book, or read it online for free.

Both [GitHub](#) and [Atlassian](#) (managing Bitbucket) have their own tutorials.

A great way to try out Git in 15 minutes is [here](#).

[Software Carpentry](#) offers intensive on-site workshops and online tutorials.

GUIs

There are several Graphical User Interfaces for [git](#).

Data Challenge

For this challenge, we will be using the paper Jaeger, Leah R., and Jeffrey T. Leek. "An estimate of the science-wise false discovery rate and application to the top medical literature." *Biostatistics* 15.1 (2013): 1-12. The paper can be found in the `readings` folder.

This tutorial has been focused on making your code reproducible and sharable with the wider world. Many journal now *require* that you submit your code and data when you publish your paper. Jaeger and Leek (2014) is one such article. While an earlier article claimed that over 80% of medical results were probably false discoveries, this work uses a slightly more sophisticated analysis of the p-values reported in the medical literature, to determine whether the observed distribution of p-values is significantly different from those drawn from a uniform distribution. Reassuringly, the authors find that the false discovery rate is lower (~20%), but still troubling, across many medical journals.

- 1) Use the published code in Jaeger and Leek (2014) to reproduce the reported mean and standard deviation of the FDR of *The Lancet* to 4 significant digits. Note that you should *not* need to scrape PubMed to do this.
- 2) Use the scraped abstract data provided in `data` and use the authors' code to compute the FDR for the journal PNAS. Find the mean and s.d. of the FDR for this journal.
- 3) Bonus question: use the authors' code to first scrape PubMed for abstracts, and next compute the FDR for the Journal of Psychiatric Neuroscience (PubMed abbreviation: J Psych Neuro). You will find a bug in the code that you will need to hunt down and fix.
- 4) Bonus question: Find the part of the code where the authors' initialize their random number generator.

Go to [this link](#) to submit your team's answers.