# Reproducibility of data analysis

*Stephanie Palmer & Stefano Allesina & Graham Smith*

## Contents



## 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 will also need GitKraken.

## `git` Glossary

- `branch`: *n.* A lineage of commits. A repository can have multiple branches, and committing changes to one will not affect the others. You can switch between branches (see `checkout`). When you switch branches, all your project's files change to the state they were in at the last commit on that branch. *v.* To create a new branch.
- `checkout`: *v.* Switch branches. Replaces all files with the files from the checked out branch.
- `clone`: *v.* Create a new local copy of a remote repository.
- `commit`: *v.* Take a snapshot of the current progress. *n.* A snapshot of the project at a particular point in time
- `init`: *v.* Create an empty repository.
- `pull`: *v.* Download the latest version of the project from the remote repository.
- `push`: *v.* Upload your latest local commit to the remote repository.
- `repository` (`repo`): *n.* Entire history of project. Set of all commits on all branches.
- `remote repository`: *n.* Repository hosted on a server (e.g. GitHub)
- `stage`: *v.* Add changes to the set that will be committed. (does not commit!)

# 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:

- back up versions of your code using `git` and GitKraken
- make your code freely accessible to the wider world via GitHub
- make your code readable
- reproduce your calculations precisely

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.

# What is reproducibility?

Here we'll consider three levels of reproducibility:

0. Accessibility
1. Replicability
2. Reusability
3. Extensibility

These are hierarchically ordered in necessity, difficulty, and goodness.

Reproducible code must be accessible to other researchers. However, accessible code is not necessarily reproducible in any sense. If the code is incomprehensible or doesn't run, then you may as well not have published anything at all.

The first real level of reproducibility is replicability: Does your code run and produce the result you said it would produce? If yes, then at least your result is replicable. If not, then you're just asking your readers to take your results on faith. Indeed, replicability is a bare minimum to qualify as "science."

Most scientists would probably aspire to reusability, the second level of reproducibility: can another scientist take your code and apply it to their own data? In order to apply your code correctly to their own data, another scientist will need to understand your code quite well. Therefore reusability requires more understandable code.

Finally, extensibility requires that your code be reconfigurable to address either different questions or different data. This is fundamentally different from the preceding, and is often beyond the scope of scientific inquiry. If you've written extensible code, you've made a tool, so this is more akin to methods development.

# Accessibility

## 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 replicate a figure you made three months ago, version control is for you (I *guarantee* you will need to do this at some point).
- 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 projects, and is essential for large collaborative projects. Without version control, for small projects like manuscripts, often researchers play a game of hot potato. They pass

the manuscript back and forth, and the manuscript accumulates an increasingly incomprehensible name (e.g. "Significanceinnovation_draft_2_gs_edits_SEP.docx", a relatively benign example). You can easily lose track of who has the latest copy, or lose time when one collaborator fails to make promised changes for weeks.

Version control keeps 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.

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

## Introduction to `git`

For this introduction, we will be using `git`, a version control system that is free and is available on all major operating systems. `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.

When you start working on a new project, you tell `git` what directory will contain the project. Then `git` takes a snapshot of that directory to create the beginning of your repository. If the project is brand new, this may be a snapshot of an empty directory. After you've done some work, you can tell `git` to take a new snapshot, called a **commit**. All snapshots remain available—you can always recover previously committed versions of files.

`git` 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. Merging is the chief advantage of `git` over some other version control systems.

**Exercise 1**: Your First Repo

1) Let's `init` your first repo from scratch! To do so, in GitKraken select `File > Init Repo`. We'll be creating a "Local Only" repository for this simple example. That just means we'll not be putting any files on any website (such as GitHub). The files are only stored on your computer. Browse for `Documents` or your equivalent. Note that you cannot put a `git` repository inside another `git` repository.

2) Now we have an empty repository. Let's put something in it. Using a text editor of your choice (Notepad, TextEdit, Sublime, even RStudio), create a file called `origin.txt` with the text "An abstract of an Essay on the Origin of Species..." and save it.

3) Now go back to GitKraken. At the top of the center panel, an entry has been added, `// WIP`. This stands for "Work In Progress" and it indicates you've made changes to your project that have not been committed to the repository. We have two steps to record the changes to the repository: First, we stage the changes. Second, we commit the staged changes.

4) Stage the change.

5) Type a commit message (e.g. "Began Essay on the Origin of Species") and commit the staged change.

6) Now let's get crazy! Create a new file called `end.txt` with the text "Some say the world will end in fire..." AND add onto `origin.txt` so that it says "An abstract of an Essay on the Origin of Species. Darwin's treatise posits that the process of natural selection passes heritable traits of more advantageous adaptations to subsequent generations. In this essay, we examine..."

7) Go back to GitKraken, and `// WIP` should be back. This time when you click on it, you should see two changes are unstaged, one for each file. One is a change, and one is an addition.

8) We'll commit them separately, since they don't really have anything to do with one another. So first stage the new file `end.txt`, type in a commit message, and commit. You should still see `// WIP` at the top, but underneath should be the commit message you just typed.

9) Now click again on `// WIP` and stage the changes to `origin.txt`. Notice when you do so that there are *three* changes to the file: the original first line was deleted, and the new first line was added. That's just how `git` works. It only does whole-line changes, not within line changes. Of course the second line was simply added.

If you use `git` in the wild, you'll find that staging and committing is 90% of what you do.

## Remote repositories

For the remainder of the tutorial, we'll be using a more complicated example based in code hosted on GitHub. So far, we have been working with a local repository, meaning the repository is hosted only on your computer. Usually, you will also want to keep a copy of the repository online, called a "remote repository". With a remote repository, you can collaborate with others, sync your code across multiple machines, and back up your code.

The most popular option for hosting remote repositories is GitHub. 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 do publish!

After the initial setup, 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 tracking a remote repository, you `pull` the most recent version from the server to sync your local copy of the project to the most recent version. When you are done working, you `push` your commits to the server so that other users can see them.

## Daily Workflow

1) Pull any new changes from your collaborators
2) Work on your project
3) When you've done something meaningful, stage your changes
4) Commit your changes, writing a meaningful commit message
5) Repeat steps 2-4
6) When you're done for the day, or when you've finished code that you want your collaborators to be able to access, push your changes to the remote repository (e.g. on GitHub).

## Clone a Repo

Just as you want to make your code reproducible for other researchers, one day you will want to use another researcher's code. When that time comes, if that researcher has politely hosted their code on GitHub, accessing their code will be as simple as this:

1) In GitKraken, `File -> Clone Repo`

2) In "Where to clone to" browse to any location on your computer that is NOT within another `git` repo. For now, I'd recommend your "Documents" folder.
3) In your browser, navigate to the repository, click the green button "Clone or download" and copy the URL that appears.
4) Paste the URL into the URL field.
5) Clone the repo!

### Help using GitKraken and `git`

GitKraken is a graphical user interface (GUI) on top of the older command-line interface (CLI) `git`. If you've never used `git` before, GitKraken has a good tutorial: https://support.gitkraken.com/start-here/guide

## Replicability

So far, we've covered making your code accessible, whether to yourself in the future (via version control) or to other researchers (via *remote* version control). But accessibility is only the first step. To go further, we need to make sure other researchers can understand our code. You'll learn more about this in defensive programming, but the basics bear repeating.

In order learn how to write understandable code, we're going to try reproducing some figures ourselves.

### Understanding the Code

Well-written code is a lot like a well-written essay: it should be understood from the top down.

At the top level, every project should have a file called simply **README**. The README introduces the project as a whole. It should explain the purpose of the project and direct the reader to important files (e.g. the script that runs the analysis). Thus the README is analogous to the essay's introduction.

Then, analogous to the introduction to a section of your essay, each **file** should begin with a few lines of comments explaining what will be found in that file. Additionally, such comments usually include the author's name so future readers know who to blame and who to contact in case of questions.

Similarly, every **block of code** should have a comment describing its purpose. A block of code is simply contiguous lines of code isolated by whitespace. Blocks are like paragraphs, so there isn't any rule as to how to make them. Loosely, if you can write a concise comment describing the action of some lines of code, that would make a good block. The comment describing the block's purpose is analogous to the topic sentence that should appear near the beginning of a paragraph (usually).

Finally, the most specific comments are **in-line comments**. As a beginning programmer, you should probably use in-line comments liberally, to make explicit to yourself what each line does. However, as you gain experience, inline comments should be used sparingly, if at all. There are a few reasons to avoid in-line comments once you gain experience coding:

1) If the code in a single line needs a comment to be understandable, then it's too complicated.
2) Similarly, if your code is written well, then the comment would be redundant.
3) Often, you'll change code but forget to change the comment, leading to misleading in-line comments.

#### Read the code

While comments are useful and necessary, you should always strive to write code that is understandable by itself. Let's look at an example. You can see a this example drawn out to a painful conclusion in `somecode.R` in the code folder. The below code is *technically* fairly simple, and shouldn't really need any comments to be understandable. But *practically* it's another story. See if you can tease out its purpose, as-is:

```
t0 = 0
a = 35000
```

```
r = 0.05
acc = 0
for (i in 1:45) {
  acc = acc * (1 + r) + a
}
print(acc)
```

This code technically "works." It will run, and it will print a result. But what does the result mean? No idea! Below are some good coding principles that it violates. We'll rewrite the code using these principles and make the code much more understandable.

- **Use good names** Use variable names that are self-explanatory. For example, `a` in the above code would more helpfully be called `annual_income`. Don't worry about using increasingly long (yet descriptive!) names. In RStudio, you can simply type `annual[TAB]` to see a list of all functions you've defined whose name starts with `year`, so the extra characters don't waste time and can only help you.
- **Do one thing at a time** When you put several things on one line, it can make the line more difficult to parse, much like a run-on sentence. Instead, separate multi-part commands into multiple lines.
- **Don't use magic numbers** This is a special case of "use self-explanatory variable names." Always name numbers by putting them into variables and using the variable in calculations. For example, the code `area <- 5 * 3` is not as clear as `area <- width * height`. In our code there's only one magic number: `45`. We should instead name this `years_to_retirement`.
- **Encapsulate code into functions** Putting code into well named functions is easily the most effective way to write self-documenting, understandable code. You'll understand the code perfectly when I tell you that we should make it a function called `calculate_retirement_savings`.

```
calculate_retirement_savings <- function(annual_income, annual_interest_rate,
                                         years_to_retirement, current_savings=0) {
  for (i in 1:years_to_retirement) {
    current_savings = current_savings * (1 + annual_interest_rate)
    current_savings = current_savings + annual_income
  }
  return(current_savings)
}
calculate_retirement_savings(annual_income=35000, annual_interest_rate=0.05,
                             years_to_retirement=45, current_savings=0)
```

We can understand this code much more readily. The code calculates retirement savings after 45 years, making some assumptions about savings and interest rates along the way.

This code is so well written, you might even notice a problem with the calculation! Most grad students can't save their entire annual income. Good programming like this is defensive because you can more easily notice problems like this. Good programming improves reproducibility because now anyone can understand the purpose, capabilities, and limitations of our code. When you re-run old code, you need to understand what's happening. If I give you a function called `calculate_retirement_savings`, you have a pretty good idea of the purpose, but that's it. Does it assume a constant interest rate? Constant income? Constant expenses? Are taxes calculated? Does the function take advantage of Roth or Traditional IRAs? Does it know to hide excessive returns in a secret account in the Caymans? These are all important considerations in planning your retirement, and you'll only know what my function does by actually reading the code, so you can only *reuse* it if you can understand it. And if you want to *extend* it, then understanding it is just the first step.

## DIY lottery

Now let's try some exercises that combine what we've covered so far. In this section, we'll clone a repository containing some bad code, we'll fix the code, and then we'll try to understand the code well enough to analyse the results.

Follow the instructions in "Clone a Repo" to clone https://github.com/grahamas/BentCoinLottery

Before you try reading the code, we should tell you that the code uses one of R's random number generator functions `runif`, which will give you uniformly distributed numbers on the interval $[0, 1]$.

> **Exercise 2**: Use meaningful variables
>
> Go through the lottery simulation, making all variable names meaningful, both by changing variable names and by introducing new variables to get rid of magic numbers.

### Probability interlude

No matter how well documented code may be, you'll still need some foundational understanding of the concepts being implemented, so that you can then understand the implementation itself. So let's pause and go over probability, which is key to understanding our lottery.

Each flip of a coin with probability, $p$, of heads is an example of a *Bernoulli trial*, which is the general 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*, which extends the Bernoulli trial to many flips.

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

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 "$P_n(k)$" represents the probability distribution of $k$ heads out of $n$ tosses. We're calling this a "bent coin lottery" because the probability of heads is not $1/2$, it's some other number. The coin we are simulating flipping isn't a fair coin, perhaps it's been subtly bent so that it mostly lands tails up.

Knowing this, you can use the function `rbinom` to generate draws from a binomial distribution. In our lottery, this would amount to flipping the coin `n_flips` times.

```
n_flips <- 3
p_heads <- 0.1
rbinom(n_flips,1,p_heads)
```

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

> **Exercise 3**: Use functions
>
> Rewrite the lottery to use `rbinom`. In case you decide to change the ticket generator in the future, abstract the ticket generation into a function called `generate_ticket`. What arguments should this function accept? What should it return? Be sure to note both answers in a comment at the beginning of the function. Does using this meaningfully-named function make the code more readable? What about using `rbinom`?

## Simulating noise

For the remainder of the exercises, we'll need a bit of extra background. First we'll discuss "seeds," which are essential to making your stochastic code reproducible. For example, seeds will make it possible to draw the exact same "random" lottery tickets as your neighbor. Second we'll go into a bit of the math underlying our lottery tickets (the binomial distribution) so that we can do some analyses of our generated data.

### 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 4**: Playing with seeds

1) Test whether or not you and your neighbor get precisely the same sequence of numbers when you use the same seed.
2) Will this work if you use different RNG algorithms?
3) Try changing your RNG algorirhm using `RNGkind` and compare your results with your neighbor, when you use the same seed.
4) 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.

**Noisy processes**

Many of the variables that we observe in biological recordings fluctuate, sometimes because we cannot control all the states of the external and internal experimental system, other times because thermal noise makes the state of the biological system we interrogate inherently variable. Examples of fluctuating quantities in biological systems include: the number of a certain type of molecule in a cell; the number of open channels in a cell; the number of electrical action potentials or "spikes" emitted by a neuron in response to a stimulus; the number of individuals in a population at a particular moment in time; the number of bacterial colonies on a plate. These are all quantities that we can make precise claims about, on average, but cannot specify with certainty for any particular experimental observation.

It is useful to model not only a mean value for a fluctuating variable, but the full shape of its distribution of values. For example, if we observe the firing of neurons in the brain to repeats of the same external stimulus, the precise times of spikes will vary between repeats. By fitting the statistics of this noise to models, we deepen our mechanistic understanding of the neural response. We can test whether or not the "noise" we observe is consistent with a truly random source of output variation, or if it has some structure that tells us about interactions between the biological components and their environment.

Often, noise in biological systems is modeled by what's called a Poisson process, whose values follow a Poisson distribution. The function you wrote to sample the bent coin lottery generated tickets whose statistics follow the binomial distribution. The binomial distribution approches the familiar Poisson distribution, in the limit of a large number of trials, $n$, or a small probability of the event, $p$, per trial:

$P_n(k)_{n\to\infty} = \frac{\lambda^k}{k!}e^{-\lambda}$

where $\lambda$ is the average rate of occurrence of our event in $n$ trials. We have just written down the Poisson distribution. You will see this used as a model for biological variability again and again, either explicitly or implicitly. It is important to think about whether or not it is a good model for the system under study each time you come across it or are deciding to use it for your own research. Notes on deriving the relationships between some common distributions are provided in the readings folder.

> **Exercise 5**: Replicate a figure
>
> Now using our knowledge of seeds and Poisson distributions, we'll try to replicate a figure created in the analysis of the lottery.
>
> 1) `checkout` the branch called "analysis."
> 2) Find the code necessary to replicate Figure 1 (hint: when you switched branches, the README was updated)
> 3) Replicate Figure 1 (hint at the bottom of "References and Readings")

# Usability

For our final exercise, we'll try to make a figure of our own. From Exercise 5, you should be on the "analysis" branch.

> **Exercise 6**: Make a new figure
>
> 1) When is the distribution modeled in Exercise 5 approximately normally distributed?
> 2) Make a figure analogous to Figure 1, but with newly generated data nicely approximated by a normal distribution.
> 3) Make sure you comment your code and include all the information necessary to reproduce your figure.

# Data Challenge

For the data challenge, you'll be analysing real data from research papers found in the `readings` folder (the data is found in the `data` folder).

**Arthropod dispersion**

In 1941 and 1942, zoologist LaMont Cole, then working at the University of Chicago, set out to survey species diversity and distribution in the woods and pastures of Kendall County, Illinois. He was particularly interested in which species co-occurred in woods versus grazing land and how their numbers varied with changes in humidity and temperature throughout the year. He laid thick oak boards in a variety of locations and counted the number of "cryptozoic" (animals found under stones, rotten logs, tree bark, etc.) individuals found under the boards several times a week, over the course of a year. For his spatial distribution studies, he aimed to determine whether arthropods distributed themselves randomly or if they had a more complex interaction pattern with each other or with their environment.

If the bugs are randomly distributed, then the counts of the bugs under the boards should be Poisson distributed. If there is some pattern, we might try to fit them by a Lagrangian Poisson Distribution or LGP, which accounts for the attraction or repulsion of individuals to each other. The LGP has a mean rate parameter, $\lambda_1$, just like in the Poisson distribution. The LGP has a second parameter, $\lambda_2$, that describes the deviation from expected Poisson dispersion.

> NB: A Poisson distribution has a variance-to-mean ratio equal to 1. Show that the variance of the spider count distribution is approximately equal to its mean.

$\lambda_2 > 0$ implies that organisms are over-dispersed (i.e. they have a variance greater than their mean number) and may be attracted to one another, forming more large-number clusters than expected by chance. $\lambda_2 < 0$ implies that organisms are under-dispersed and are likely repulsed by one another, resulting in a more even distribution across space and, hence, lower count variance. The LGP distribution, in terms of these two paramters, is

$P(k) = \lambda_1(\lambda_1 + k\lambda_2)^{k-1} \frac{e^{-(\lambda_1 + k\lambda_2)}}{k!}$.

When $\lambda_2 = 0$, we get precisely the Poisson distribution.

**The challenge**

Check visually how the distributions of the three bug types (for weevils, its their eggs and they lay their eggs inside beans, so it's eggs per bean instead of bugs per board, but you will plot and fit the data in the same way) fit to a Poisson or the LGP distributions.

1) Plot the Poisson distribution with the same mean as the spider counts, along with the data
2) Plot the Poisson distribution with the same mean as the sowbug counts, along with the data
3) Plot the Poisson distribution with the same mean as the weevil egg counts, along with the data
4) Add a curve to Plot 1) showing the LGP distribution with the parameter hint below for the spider counts
5) Add a curve to Plot 2) showing the LGP distribution with the parameter hint below for the sowbug counts
6) Add a curve to Plot 3) showing the LGP distribution with the parameter hint below for the weevil egg counts

Each figure should have datapoints corresponding to empirical data, and lines corresponding to theoretical distribution. Indicate in the README how well the each distribution fits these data sets, as well as the chosen $\lambda_2$.

For LGP, you can use the function `dLGP` from the library `RMKdiscrete`.

For spiders, you can set $\lambda_2$ to 0 and $\lambda_1$ to the mean.

For sowbugs, you can use the empirically determined $\lambda_2 = 0.53214$ and estimate $\lambda_1$ by the formula $\lambda_1 = \text{mean} * (1 - \lambda_2)$.

For weevil eggs, look at the data to see if you can gauge their sociability (i.e. do weevils lay their eggs all together or do they seem to intentionally separate them?) and thereby start guessing appropriate $\lambda_2$.

### The requirements

1) Create a **private** repository on GitHub.
    - On the GitHub homepage, select "New Repository" and then on the next page select "Private"
2) Add all members of your group as collaborators, plus Graham (username: grahamas) and Stephanie (username: sepalmer).
    - On your repository's GitHub page, select "Settings -> Collaborators" then add by GitHub username
3) Every member of your group must make at least one substantive commit.
4) Follow the commenting guidelines outlined above.
5) Try to write functions, rather than copying and pasting code.

## 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.

### Hint to Exercise 5

- You'll need to go through the commit history.