Analytical Workflows

All course notes strung together

September 8, 2020

A quick and dirty intro to project structure, coding, and version control workflows

September 8, 2020

In what follows we'll talk about three things:

- How to structure your project(s)
- Using `git` and `github.com` for version control
- Best practices in coding

Contents

Project structure

The general recommendations of this section in regards to establishing a consistent structure for your project(s) should apply whether or not you plan to use version control software to manage your project(s) or not. For example, the recommendations apply equally if you plan to use `Dropbox` or the equivalent (which you should *most definitely* be using if you're not going to use version control software).

Like many grad students, I finished my thesis with

- one master folder called `Data` containing a bunch of sub-folders within it containing all the various data sets (mostly Excel and CSV files) and databases (mostly MS Access) that I'd collected or collated over the years;
- one master folder called `Rcodes` containing a bunch of sub-folders within it (each with a different project or analysis of my data);
- one master folder called `Mathematica` that similarly contained a bunch of sub-folders;
- one master folder called `Manuscripts` that contained all the papers and chapters I'd attempted, completed or published;

and a bunch more similar folders all variously named within my overall `Research` folder. You might currently have something similar for just your Thesis.

Turns out that's a poor way to organize your work for a variety of reasons, including reproducability (by yourself down the road or someone else if you managed to pull all the necessary parts together for that person); the ease and efficiency with which you might expand, modify or branch off of prior work; and the ease of performing data and code backups.

I now organize my work using a *project mindset*. I don't do this for each and every project idea or analysis I try out, but I do use it for "definitely doing this" (i.e. planned out paper (thesis chapter)) and collaborative projects. By *project mindset*, I mean that everything associated with a given project is contained in one folder. I still use a combination of `Dropbox` and `git` to organize my projects (and you should *never* put a `git` folder within your

`Dropbox` folder, or vice versa), but within each of either `Dropbox` or `git` I have all my project folders organized within a master `Projects` folder.

That said, defining "a project" can get difficult (esp. within your thesis work), so a fair bit of forethought can be needed. It's not trivial.

Ben says: I also use *project mindset* to organize my folders. Something I like about project mindset is that it encourages what you might call deliverables-based thinking. By identifying and naming the "definitely doing this" projects, I am encouraged to consider my priorities, both within and among projects.

For each project I am forced to think clearly about what it is about by needing to name the folder. Asking "what should this project folder (or Git repo) be called?" (and insisiting on an *informative name*, eg not "Mark-Bencoursething") is pretty close to asking "what is this project about?" So project organization supports clear scientific thinking.

I find project mindset also promotes better time management. For instance, all my project folders are contained within three superfolders: Active, Complete and Archive. The folders within Complete are named with dates and brief titles of publication. The Active folder contains stuff I am working on right now, that has not "shipped" yet. Archive is for stuff that is on the back burner.

In this setup, the project folders within the Active folder, each with a name that reminds me of the objective for that project, becomes a kind of high-level to-do list. The goal is to be able to one day drag those folders from Active to Complete. Crucially, if the Active folder gets too full, I know I will not be able to do that because my attention has become too divided. So then I ask myself which are the most important few projects to me, and drag the rest to the Archive folder. It's not that I can't do them later. It is just recognizing that a) they are not done yet, and b) they are not the first, second or even third priority. If a) and b) are true, into the Archive folder they go!

OK, back to Mark, and the structure of an individual project folder.

Your Project folder / Respository

Within each project folder I usually have the following sub-folders:

`data`	The original (and cleaned) data required for the project
`code`	All the scripts needed to perform the analyses
`output`	The results of the analyses
`figures`	The final figures (and tables) that go into the manuscript
`manuscript`	Manuscript(s) derived from the project
`pdfs`	(*as appropriate*) Collection of relevant papers, manuals, etc.
`biblio`	(*as appropriate*) Bibliographic files
`talks`	Presentations you've given on the project

The last two (`pdfs` and `biblio`) sometimes get put in sub-folders within the `manuscript` folder. The `manuscript` folder often contains sub-folders, one for each journal I've submitted to. The `figures` folder sometimes gets put within each of the `manuscript` sub-folders, depending on how much I decide to change what the final figures are for different journals. The contents of `figures` differs from the rough-and-dirty figures I save into the `output` folder. Sometimes `tables` get their own folder (if there are a lot of them).

Ben says: My sub-folder structure is similar to what is listed below, with variations depending on the project and preferences. For example, my reference manager of choice keeps all my pdfs in one place, so instead of having a pdfs folder in each project folder, I have 'folders' for each project within my reference manager. Either way, the same goal is achieved: a logical hierarchical structure that makes it easy to find and keep the various pieces of a project. Back to you Mark.

Most of the time when using `git` you'll have one *repository* associated with each one of your *projects*. A *repository* as thus synonymous with a *project folder*. When using `git` you'll also have a few other files within the repository: a `README.md` file and a `.gitignore` file. If you're using `R-Studio` in combination with `git` (as we will below), then you'll also have an `.Rproj` file in the repository.

Working with `git`

What is version control?

Version control software allows you to save and provide meta-information on any and all changes you make to a set of files and directories. It allows you to more easily document changes and the evolution of your files, find and correct bugs that have crept into your code, and revert back to old file versions when needed. By using version control you'll now what, when and (hopefully) why changes were made. If you're using the software collaboratively, you'll also know by whom changes were made. Backing-up is virtually implicit (though not totally so), especially when you also use a networked (off-site) server as a host for your repository.

We'll be using `git` as our version control software. There are others out there (e.g., `Subversion`). We'll also be using `Github` as our host. There are others out there (e.g., `Bitbucket`).

git

`git` was developed by Linus Torvalds (the "Linu" in Linux). Most of its features are accessed by command-line and are intended for large-scale collaborative programming and software development purposes. Of course, we'll be using `git` for much smaller, specific projects, thus we won't bother with many of these feature. We'll also interact with `git` using GUIs (e.g., `R-Studio`) rather than command-line.

Required reading to get an overview of how `git` works:
`https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F`.

Github

`Git` enables you to store a complete copy of the project on your local machine, including its history and all versions. That means that no centralized server is necessary. However, if you want to collaborate with others, have multiple computers with which you'd like to work, or want to create true back-ups

of your project, then you'll also want to use a server on which to host your repository. Fortunately, there are a number of free providers.

For `git`, the primary options are `Github` and `Bitbucket`. The former is more developed (more bells and whistles), is currently more widely used, and is perhaps a little easier to work with. The two don't differ all that much except in one regard: the number of free versus public repositories. While `Github` has a limit on the number of private repositories, `Bitbucket` has a limit on the number of collaborative projects (having more than 5 collaborators). (There may still be perks regarding the number of repositories when you sign up using an academic email address. See `https://help.github.com/en/articles/applying-for-an-educator-or-researcher-discount`)

To do: Create an account on `github`. (I suggest creating accounts with both hosts. I use them for different projects, as needed.)

Installing and configuring Git

To install and configure `git` on a PC, go see
`https://git-scm.com/book/en/v2/Getting-Started-Installing-Git`

To install `git` on a Mac:

`http://code.google.com/p/git-osx-installer`
or
`http://git-scm.com/downloads`

After you install `git` there's a little (minimum of) command-line configuration to perform. On a Mac, open a `Terminal` window and type in the following:

```
$ git config --global user.name "Mark Novak"
$ git config --global user.email "Mark.Novak@oregonstate.edu"
```

Some of you might want to set up an editor to be used in git, for example:

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

You can check to ensure that these commands when through and see what other things you might want to configure using

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

For more, see
https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup

Repository setup

There are command-line methods for doing everything we're going to do below. Indeed, command-line is the default way that most people interact with `git`. (See last page for a cheat sheet.) Instead, we're going to make use of the tools made available through `Github`, starting with the very first step of initiating a repository and getting some minimal information associated with it. That is, rather than creating and setting up a repository on our computer by command-line and then connecting it to `Github`, we'll create and set it up on `Github` and clone it to our local master folder of projects.

Simply log in to your `Github` account, click on "New Repository", and follow the instructions. These should include options for private versus public (pick the latter for this class), initializing with a `README` file (which you *should* do), and adding a `.gitignore` file (which you *should* also do).

The `README` file is the first file that anyone will look at when they inspect your repository (assuming it's public). It should give an overview of what the project is about and what the various parts of the project structure are. You can edit it at any time, but for now put some minimally useful information in it. The `.gitignore` file contains a list of all the files that you want `git` to ignore (not monitor for changes). Selecting `R` from the dropdown list will auto-fill a bunch of it for you. You typically won't do much with this list, though it is useful sometimes. (For example, you may wish to have `git` ignore (not manage) the `.Rproj` file that `R-Studio` creates.)

You should now see a new webpage – your Repo page – that shows you what's in your repository. For now it contains only the `.gitignore` and `README.md` files, the latter of which has its contents displayed. As I said earlier, there's a lot of bells-and-whistles at your fingertips here (the most useful of which for collaborative projects is the `Issues` feature). You *could* start dragging-in directories and files to add them to your repository, but we're *not* going to do that. Instead, we're going to `clone` this repository to our local machine, then add our various project sub-folders to it (from Your Project folder / Respository) and go from there.

To clone the repository, click the green `Clone or download` button and copy the provided URL.

Now there's a couple ways to clone your repository to your local machine. The preferred method depends on how you're most likely going to interface with `git`. You could:

1. use command-line to clone. Open `Terminal`, `cd` into your `Projects` master folder, then type `git clone` followed by the URL;
2. use a visual `git` GUI client to clone the repo; or, if you're going to be primarily using `R` via `R-Studio` anway,
3. set up a "project" within `R-Studio` first and provide it with the URL during setup. It'll then clone the repo for you.

R-Studio and Git GUIs

I use `git` for both `R` and non-`R` (e.g., `Mathematica`)-based projects. Only `R-Studio` has integrated `git` functionality, so I use a visual `git` GUI client (e.g., `Sourcetree`) for some projects because I can't be bothered to memorize the `git` command-line commands. Since most of you are probably using `R`, it's probably worthwhile to use `R-Studio`'s `git` integration feature (which is the only reason I use `R-Studio` to begin with).

You'll first need to tell `R-Studio` that you have `git` installed, so go to its Preferences, select `Git/SVN` and fill in the details: either click on the Help link or go to `http://r-pkgs.had.co.nz/git.html` to see what to do.

Now create your "project" within `R-Studio` by selecting "New Project". Select `Version Control: Checkout a project from...repository`, select `git`, and fill in the details including the URL you got from `Github`. The directory in which you place your repository should be your master folder. `R-Studio` will "restart" and then you'll be in your project (as evidenced by its name appearing in the top-right of the interface). Clicking on the `Files` tab will show you what's in the repository (which should at present be: `README.md`, `.gitignore` and the newly created `.Rproj` file).

You may now create (or move in) all your project sub-folders (see Your Project folder / Respository).

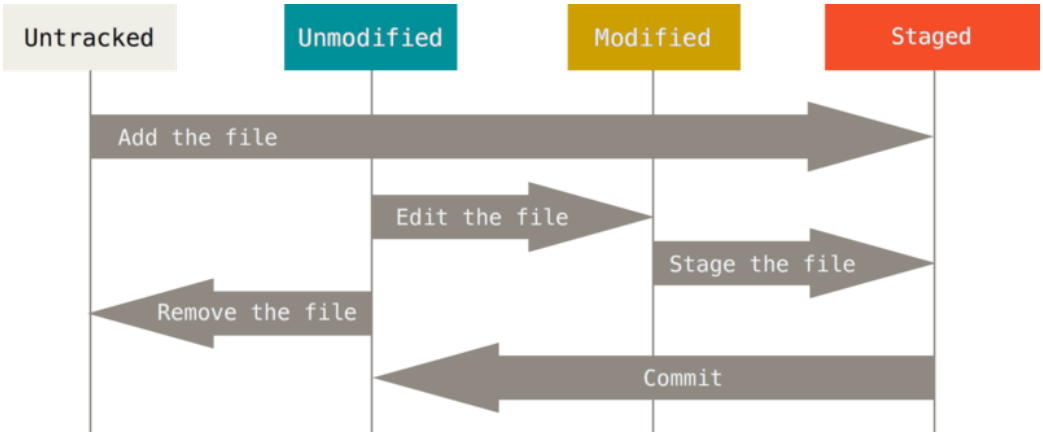

Figure 1: The `git` life-cycle.

Git work flow

First, **required reading:**
`https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F`

Basically, files (or directories) exist in one of four states of a life-cycle: untracked, staged, unmodified, or modified (see Fig. 1). The standard workflow is thus:

1. Add or modify some files.
2. Stage the new or modified files.
3. Commit the changes (moving them from the Staging Area to the "memory" of the repository).
4. Repeat.

The motto of `git` is "*commit early, commit often*". Every time you add or remove something from your project (i.e. a file or a chunk of code), you want to commit those changes. Ideally, each commit should correspond to a "logical unit", one that you are able to describe in a few words (e.g., "function to perform resampling added to analysis script"). Choosing when to commit is quite important, especially when you're debugging code: For example, if you've discovered your code has two bugs then you should commit each one of the fixes seperately, not together. That way you can undo either fix

Figure 2: Don't let this happen! (source: `http://xkcd.com/1296/`)

independently if, for example, you messed up in one of your fixes or your fix created a different bug somewhere else in your code.

Within `R-Studio` (and assuming you have your `R-Studio` project opened), looking at the `git` tab will show you a list of all the files (and directories) that have been changed, removed or added to the project since the last time you committed. Clicking on the check-boxes associated with each file will add them to the staging area. Clicking on `Diff` or `Commit` will open up a new interface (the staging area). In the top-left corner you'll see a list of the staged files. Selecting one of the files will bring its contents up in the window below which highlights the text that has been added (in green) and removed (in red). Select all the files you want to commit, then add a commit message to the top-right window, and commit.

You've now updated your local respository. Clicking on `History` (top-left) will show you all your past commits. Remember, "*commit early, commit often*".

The final thing to do (often, for back-up reasons, but not necessarily following each commit) is to `Push` your commit to `Github`. `Pull` obviously does the opposite: bringing commits that have been saved to `Github` (by others, or by you on a different machine) to your local machine.

Note: *To avoid conflicts, the safe thing to do is to always pull before you start making edits, commit and push. There are of course ways of dealing with conflicts and merging files (that will arise if two people are working on the same file, for example), but for our mostly single-user purposes it's easier to just avoid them.*

Git resources

There's a whole lot more functionality to `git`, some of which is important and very useful (such as resolving conflicts, branching and merging, and reverting back to old versions), but we're not going to get into that unless we have extra time. For more information, check out the following resources:

`https://www.codecademy.com/learn/learn-git`
`https://www.atlassian.com/git/tutorials`
`http://rogerdudler.github.io/git-guide/`
`https://git-scm.com/book/en/v2`

Coding

The reality is that for any non-trivial task, most of your time will be spent de-bugging your code, not writing it. There are two over-arching principles that will, when practiced, greatly increase your efficiency: writing *modular* code and writing *clean* code. In class we'll discuss modularity first (it relates to the *project mindset* referred to in the context of `git`) and will then talk about code-writing best practices (styles guides).

Principle of Modularity

If your experience learning to code as been anything like mine then, either when you first started coding or when you wrote homework assignments for Stats class, the code you created for a given project probably consisted of one long giant file. If you learned to use `Markdown` or `Sweave`, that code probably had some number of sections within it, just like you might write a paper or thesis chapter (Intro, Methods, Results).

That'll work fine for small (tiny) projects or homework reports, but probably not for anything the size of a paper or thesis chapter that contains some combination or subset of data, analyses (whether statistical or non-statistical modeling), and data- or results visualization.

Thus, as alluded to in Your Project folder / Respository, you should give your project a useful structure:

Your `data` folder should contain all the data you need for the project. Most of the time, this will include both the *original* unedited raw data files (regardless of their format) as well as *derived* data files (that you have used code to produce). No files should be duplicates or derived copies of each other (see Fig. 3); remember: versions will be tracked by `git`.

Your `code` folder contains your various script files. For anything but the smallest projects there should be multiple files that contain scripts for performing different tasks. For example, you might have:

`data_prep.R`	Script that pulls in the original data, pre-processes them (rearranges, correct errors, standardizes names, splits up or merges different data sets), and spits them back out in a "clean" format needed by your subsequent analyses
`my_functions.R`	Script containing the functions you have self-defined to perform your analyses
`analysis.R`	Script that sucsk in your "clean" data, performs your analyses (using built-in, self-defined and package functions), makes some quick-and-dirty figures along the way, and exports the results to `output`
`final_figs.R`	Script that sucks in the results of the analyses to produce the final figures (or tables) for your manuscript

In general, you don't want any file to become unwieldy. Thus you will likely have multiple scripts within each of the above categories (e.g., one for each of several different data set types, one for each of several different analyses or analysis steps, or one for each of several different self-defined functions) – each appropriately named (see Code style guides). If you end up with a whole lot of scripts to perform an analysis from start to finish, then you may want one additional `RunMe.R` script that sources each of the other scripts in the appropriate order.

The key utilities of separating out everything as specified above are (1) *readability* and (2) *unit-testing*. Readability means that it's easy for anyone (including you in 1 years time) to figure out where things are being pulled from and where they're going, and no individual script is overwhelming to look through. The idea behind unit-testing is to write independent tests for the smallest units of your code. For example, whenever you write a function, you also write an associated small piece of code that tests it (before the function is used in your primary analysis). That way, whenever you modify your code or function(s), it's easy to ensure that everything is still returning the correct output.

Working directory and directory access

In order to employ the principle of modularity you need to know how to navigate between your various folders and scripts. Within a project, for example, you have a choice between using the repository folder as your working

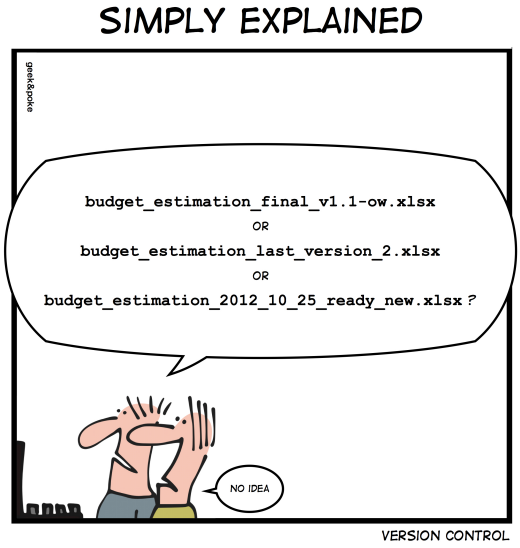

Figure 3: Never again should your data files look like this!

directory or setting your working directory to be the directory in which you have saved your scripts (e.g., your RunMe script). Your scripts will therefore need to access data and source scripts that are located in other folders. You want to set these locations as generically as possible (so that you can move your entire project to a different location without destroying the workflow, for example). You never want to set the working directory more than once, or hard-write the locations of your data or other scripts within a given script.

You can do this by specifying locations relationally using '`/.../file.R`', for example. Each repetition of '`/.../`' will move you up one level in the folder structure. For example, assuming your working directory is the `code` directory, rather than specifying the location of your data as

```
read.csv(file='C://MyDocuments/Git/MyProject/data/data.csv')
```

you should instead use

```
read.csv(file='.../data/data.csv').
```

The latter takes you up one level (out of `code` into your general `repository` folder) then into the `data` folder and to your data file. To pull in scripts, use

```
source(file='my_functions.R')
```

(assuming your working directory is the `code` directory).

Note: Macs (Unix) and Windows (Dos) use forwardslashes and backslashes differently. Macs use '`/.../.../file.R`' (forwardslashes) while Windows use '`\...\...\file.R`' (backslashes).

Code style guides

There are a lot of summarized sets of recommended best-practices for coding in general and for `R` as well. These includes aspects relating to object naming conventions (for filenames, function names, and variable names), syntax and grammar (spacing, indentation, etc.), and code structure. I won't repeat everything here, but we will go over and discuss the big ones in class. For our course, the **required reading** is:

`https://google.github.io/styleguide/Rguide.xml`

Other guides that are well worth reading for some additional suggestions and explanations are:

`https://www.r-bloggers.com/r-code-best-practices/`

and

`http://adv-r.had.co.nz/Style.html`

Within-script organization

In additional the best practices in the naming of objects and the syntax of your code, there's also an important aspect of within-code organization. Your code should consist of the following sections, each visually separated from the others:

1. Start each file with a preface that describes what it contains and how it fits into the project. You might also want to include who wrote it and when.
2. Load all required packages
3. Source required scripts (e.g., `my_functions.R`)

4. Load (or source) required data (or `data_prep.R` scripts)
5. Section(s) for major parts of your analyses
6. Export results section(s)

The last two may consist of just two sections or may have export parts immediately following an analysis. Wherever possible and appropriate, clear your workspace (`rm(list=ls())`). For example, you'd most definitely do this at the top of your `RunMe.R` script and perhaps to at the top of your `analysis.R` script (since your "cleaned" data was saved and can be reloaded, leaving all the temporary variables unnecessary), but you wouldn't put `rm(list=ls())` at the head of your `my_functions.R` script.

Thus a script file might look as follows:

```
1  ##########################################################
2  # simulateLV.R
3  # Simulates the dynamics of a predator and prey
       population according to the Lotka-Volterra model.
4  # The data produced will subsequently be used to test
       the performance of several population dynamic model-
       fitting routines.
5  ##########################################################
6  rm(list=ls()) # clear workspace
7
8  ###############################
9  # Load librairies
10 ###############################
11 library(deSolve)
12
13 ###############################
14 # Source files
15 ###############################
16 # None needed
17
18 ###############################
19 # Define model
20 ###############################
21 LVmod <- function(Time, State, Pars) {
```

```
22     with(as.list(c(State, Pars)), {
23        Ingestion    <- rIng  * Prey * Predator
24        GrowthPrey   <- rGrow * Prey * (1 - Prey/K)
25        MortPredator <- rMort * Predator
26
27        dPrey        <- GrowthPrey - Ingestion
28        dPredator    <- Ingestion * assEff - MortPredator
29
30        return(list(c(dPrey, dPredator)))
31        })
32     }
33
34  ################################
35  # Define parameters
36  ################################
37  pars <- c( rIng   = 0.2,    # /day, rate of ingestion
38             rGrow  = 1.0,    # /day, prey growth rate
39             rMort  = 0.2 ,   # /day, predator mortality
40             assEff = 0.5,    # assimilation efficiency
41             K      = 10)     # mmol/m3, carrying capacity
42
43  ################################
44  # Simulate model
45  ################################
46  yini  <- c(Prey = 1, Predator = 2)
47  times <- seq(0, 200, by = 1)
48  out   <- ode(yini, times, LVmod, pars)
49  summary(out)
50
51  ################################
52  # Plot and export data
53  ################################
54  plot(out)
55  write.csv(out, file='.../output/LV_out.csv')
56
57  ##############################################################
```

When defining a function, you should provide additional information that describes what the function does (in general terms), what inputs it takes, and what output it returns. After the function is defined it's worth adding a test case (commented out). [Note: some would separate out test cases, leaving only the function definition in its own script.] For example:

```
1  ############################################################
2  # Function to add stochastic noise to a time-series of
       population sizes.
3  ############################################################
4  # Input:
5  #   x -- a time series of population sizes (vector)
6  #   error_model -- gaussian (currently implemented model,
        default)
7  #   sd -- the standard deviation of gaussian errors (
       default=0)
8  # Returns:
9  #   Vector of length equal to the input vector of time
       series
10
11 add_error <- function(x, error_model='gaussian', sd=0){
12     error_model <- match.arg(error_model)
13     if(error_model=='gaussian'){
14         out <- x + rnorm(length(x), sd)
15     }else{
16         out <- x
17         warning(''Original time series returned'')
18     }
19     return(out)
20 }
21
22 # Test:
23 # source('simulateLV.R')
24 # new <- add.error(out, sd=1)
25 # par(mfrow=c(1,2))
26 #   plot(out)
27 #   plot(new)
```

When writing code, though shalt not...

- Copy-paste-edit (i.e. reuse and modify) the same code to repeatedly apply it to different subsets of data, etc.. Instead, you will convert your code to function(s) that can be applied to these data subsets
- Use `attach()` and `detach()` on your data. Instead, be explicit in naming and accessing data.frame columns.
- Repeatedly reset your working directory. Instead, use relational sourcing.
- Save your R workspace for later reuse. Instead, use `rm(list=ls())` wherever possible and appropriate.
- create large tables by hand. If you find yourself having to export large tables a lot, learn LaTeXand export them instead

git cheat sheet

learn more about git the simple way at *rogerdudler.github.com/git-guide/*
cheat sheet created by Nina Jaeschke of *ninagrafik.com*

create & clone

create new repository	*git init*
clone local repository	*git clone /path/to/repository*
clone remote repository	*git clone username@host:/path/to/repository*

add & remove

add changes to INDEX	*git add <filename>*
add all changes to INDEX	*git add **
remove/delete	*git rm <filename>*

commit & synchronize

commit changes	*git commit -m "Commit message"*
push changes to remote repository	*git push origin master*
connect local repository to remote repository	*git remote add origin <server>*
update local repository with remote changes	*git pull*

branches

create new branch	*git checkout -b <branch>* *e.g. git checkout -b feature_x*
switch to master branch	*git checkout master*
delete branch	*git branch -d <branch>*
push branch to remote repository	*git push origin <branch>*

merge

merge changes from another branch	*git merge <branch>*
view changes between two branches	*git diff <source_branch> <target_branch>* *e.g. git diff feature_x feature_y*

tagging

create tag	*git tag <tag> <commit ID>* *e.g. git tag 1.0.0 1b2e1d63ff*
get commit IDs	*git log*

restore

replace working copy with latest from HEAD	*git checkout -- <filename>*

A quick and dirty intro to project structure, coding, and version control workflows

September 8, 2020

In what follows we'll talk about three things:

- How to structure your project(s)
- Using `git` and `github.com` for version control
- Best practices in coding

Contents

Project structure

The general recommendations of this section in regards to establishing a consistent structure for your project(s) should apply whether or not you plan to use version control software to manage your project(s) or not. For example, the recommendations apply equally if you plan to use `Dropbox` or the equivalent (which you should *most definitely* be using if you're not going to use version control software).

Like many grad students, I finished my thesis with

- one master folder called `Data` containing a bunch of sub-folders within it containing all the various data sets (mostly Excel and CSV files) and databases (mostly MS Access) that I'd collected or collated over the years;
- one master folder called `Rcodes` containing a bunch of sub-folders within it (each with a different project or analysis of my data);
- one master folder called `Mathematica` that similarly contained a bunch of sub-folders;
- one master folder called `Manuscripts` that contained all the papers and chapters I'd attempted, completed or published;

and a bunch more similar folders all variously named within my overall `Research` folder. You might currently have something similar for just your Thesis.

Turns out that's a poor way to organize your work for a variety of reasons, including reproducability (by yourself down the road or someone else if you managed to pull all the necessary parts together for that person); the ease and efficiency with which you might expand, modify or branch off of prior work; and the ease of performing data and code backups.

I now organize my work using a *project mindset*. I don't do this for each and every project idea or analysis I try out, but I do use it for "definitely doing this" (i.e. planned out paper (thesis chapter)) and collaborative projects. By *project mindset*, I mean that everything associated with a given project is contained in one folder. I still use a combination of `Dropbox` and `git` to organize my projects (and you should *never* put a `git` folder within your

`Dropbox` folder, or vice versa), but within each of either `Dropbox` or `git` I have all my project folders organized within a master `Projects` folder.

That said, defining "a project" can get difficult (esp. within your thesis work), so a fair bit of forethought can be needed. It's not trivial.

Ben says: I also use *project mindset* to organize my folders. Something I like about project mindset is that it encourages what you might call deliverables-based thinking. By identifying and naming the "definitely doing this" projects, I am encouraged to consider my priorities, both within and among projects.

For each project I am forced to think clearly about what it is about by needing to name the folder. Asking "what should this project folder (or Git repo) be called?" (and insisiting on an *informative name*, eg not "Mark-Bencoursething") is pretty close to asking "what is this project about?" So project organization supports clear scientific thinking.

I find project mindset also promotes better time management. For instance, all my project folders are contained within three superfolders: Active, Complete and Archive. The folders within Complete are named with dates and brief titles of publication. The Active folder contains stuff I am working on right now, that has not "shipped" yet. Archive is for stuff that is on the back burner.

In this setup, the project folders within the Active folder, each with a name that reminds me of the objective for that project, becomes a kind of high-level to-do list. The goal is to be able to one day drag those folders from Active to Complete. Crucially, if the Active folder gets too full, I know I will not be able to do that because my attention has become too divided. So then I ask myself which are the most important few projects to me, and drag the rest to the Archive folder. It's not that I can't do them later. It is just recognizing that a) they are not done yet, and b) they are not the first, second or even third priority. If a) and b) are true, into the Archive folder they go!

OK, back to Mark, and the structure of an individual project folder.

Your Project folder / Respository

Within each project folder I usually have the following sub-folders:

`data`	The original (and cleaned) data required for the project
`code`	All the scripts needed to perform the analyses
`output`	The results of the analyses
`figures`	The final figures (and tables) that go into the manuscript
`manuscript`	Manuscript(s) derived from the project
`pdfs`	(*as appropriate*) Collection of relevant papers, manuals, etc.
`biblio`	(*as appropriate*) Bibliographic files
`talks`	Presentations you've given on the project

The last two (`pdfs` and `biblio`) sometimes get put in sub-folders within the `manuscript` folder. The `manuscript` folder often contains sub-folders, one for each journal I've submitted to. The `figures` folder sometimes gets put within each of the `manuscript` sub-folders, depending on how much I decide to change what the final figures are for different journals. The contents of `figures` differs from the rough-and-dirty figures I save into the `output` folder. Sometimes `tables` get their own folder (if there are a lot of them).

Ben says: My sub-folder structure is similar to what is listed below, with variations depending on the project and preferences. For example, my reference manager of choice keeps all my pdfs in one place, so instead of having a pdfs folder in each project folder, I have 'folders' for each project within my reference manager. Either way, the same goal is achieved: a logical hierarchical structure that makes it easy to find and keep the various pieces of a project. Back to you Mark.

Most of the time when using `git` you'll have one *repository* associated with each one of your *projects*. A *repository* as thus synonymous with a *project folder*. When using `git` you'll also have a few other files within the repository: a `README.md` file and a `.gitignore` file. If you're using `R-Studio` in combination with `git` (as we will below), then you'll also have an `.Rproj` file in the repository.

Working with `git`

What is version control?

Version control software allows you to save and provide meta-information on any and all changes you make to a set of files and directories. It allows you to more easily document changes and the evolution of your files, find and correct bugs that have crept into your code, and revert back to old file versions when needed. By using version control you'll now what, when and (hopefully) why changes were made. If you're using the software collaboratively, you'll also know by whom changes were made. Backing-up is virtually implicit (though not totally so), especially when you also use a networked (off-site) server as a host for your repository.

We'll be using `git` as our version control software. There are others out there (e.g., `Subversion`). We'll also be using `Github` as our host. There are others out there (e.g., `Bitbucket`).

git

`git` was developed by Linus Torvalds (the "Linu" in Linux). Most of its features are accessed by command-line and are intended for large-scale collaborative programming and software development purposes. Of course, we'll be using `git` for much smaller, specific projects, thus we won't bother with many of these feature. We'll also interact with `git` using GUIs (e.g., `R-Studio`) rather than command-line.

Required reading to get an overview of how `git` works:
`https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F`.

Github

`Git` enables you to store a complete copy of the project on your local machine, including its history and all versions. That means that no centralized server is necessary. However, if you want to collaborate with others, have multiple computers with which you'd like to work, or want to create true back-ups

of your project, then you'll also want to use a server on which to host your repository. Fortunately, there are a number of free providers.

For `git`, the primary options are `Github` and `Bitbucket`. The former is more developed (more bells and whistles), is currently more widely used, and is perhaps a little easier to work with. The two don't differ all that much except in one regard: the number of free versus public repositories. While `Github` has a limit on the number of private repositories, `Bitbucket` has a limit on the number of collaborative projects (having more than 5 collaborators). (There may still be perks regarding the number of repositories when you sign up using an academic email address. See `https://help.github.com/en/articles/applying-for-an-educator-or-researcher-discount`)

To do: Create an account on `github`. (I suggest creating accounts with both hosts. I use them for different projects, as needed.)

Installing and configuring Git

To install and configure `git` on a PC, go see
`https://git-scm.com/book/en/v2/Getting-Started-Installing-Git`

To install `git` on a Mac:

`http://code.google.com/p/git-osx-installer`
or
`http://git-scm.com/downloads`

After you install `git` there's a little (minimum of) command-line configuration to perform. On a Mac, open a `Terminal` window and type in the following:

```
$ git config --global user.name "Mark Novak"
$ git config --global user.email "Mark.Novak@oregonstate.edu"
```

Some of you might want to set up an editor to be used in git, for example:

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

You can check to ensure that these commands when through and see what other things you might want to configure using

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

For more, see
https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup

Repository setup

There are command-line methods for doing everything we're going to do below. Indeed, command-line is the default way that most people interact with `git`. (See last page for a cheat sheet.) Instead, we're going to make use of the tools made available through `Github`, starting with the very first step of initiating a repository and getting some minimal information associated with it. That is, rather than creating and setting up a repository on our computer by command-line and then connecting it to `Github`, we'll create and set it up on `Github` and clone it to our local master folder of projects.

Simply log in to your `Github` account, click on "New Repository", and follow the instructions. These should include options for private versus public (pick the latter for this class), initializing with a `README` file (which you *should* do), and adding a `.gitignore` file (which you *should* also do).

The `README` file is the first file that anyone will look at when they inspect your repository (assuming it's public). It should give an overview of what the project is about and what the various parts of the project structure are. You can edit it at any time, but for now put some minimally useful information in it. The `.gitignore` file contains a list of all the files that you want `git` to ignore (not monitor for changes). Selecting `R` from the dropdown list will auto-fill a bunch of it for you. You typically won't do much with this list, though it is useful sometimes. (For example, you may wish to have `git` ignore (not manage) the `.Rproj` file that `R-Studio` creates.)

You should now see a new webpage – your Repo page – that shows you what's in your repository. For now it contains only the `.gitignore` and `README.md` files, the latter of which has its contents displayed. As I said earlier, there's a lot of bells-and-whistles at your fingertips here (the most useful of which for collaborative projects is the `Issues` feature). You *could* start dragging-in directories and files to add them to your repository, but we're *not* going to do that. Instead, we're going to `clone` this repository to our local machine, then add our various project sub-folders to it (from Your Project folder / Respository) and go from there.

To clone the repository, click the green `Clone or download` button and copy the provided URL.

Now there's a couple ways to clone your repository to your local machine. The preferred method depends on how you're most likely going to interface with `git`. You could:

1. use command-line to clone. Open `Terminal`, `cd` into your `Projects` master folder, then type `git clone` followed by the URL;
2. use a visual `git` GUI client to clone the repo; or, if you're going to be primarily using `R` via `R-Studio` anway,
3. set up a "project" within `R-Studio` first and provide it with the URL during setup. It'll then clone the repo for you.

R-Studio and Git GUIs

I use `git` for both `R` and non-`R` (e.g., `Mathematica`)-based projects. Only `R-Studio` has integrated `git` functionality, so I use a visual `git` GUI client (e.g., `Sourcetree`) for some projects because I can't be bothered to memorize the `git` command-line commands. Since most of you are probably using `R`, it's probably worthwhile to use `R-Studio`'s `git` integration feature (which is the only reason I use `R-Studio` to begin with).

You'll first need to tell `R-Studio` that you have `git` installed, so go to its Preferences, select `Git/SVN` and fill in the details: either click on the Help link or go to `http://r-pkgs.had.co.nz/git.html` to see what to do.

Now create your "project" within `R-Studio` by selecting "New Project". Select `Version Control: Checkout a project from...repository`, select `git`, and fill in the details including the URL you got from `Github`. The directory in which you place your repository should be your master folder. `R-Studio` will "restart" and then you'll be in your project (as evidenced by its name appearing in the top-right of the interface). Clicking on the `Files` tab will show you what's in the repository (which should at present be: `README.md`, `.gitignore` and the newly created `.Rproj` file).

You may now create (or move in) all your project sub-folders (see Your Project folder / Respository).

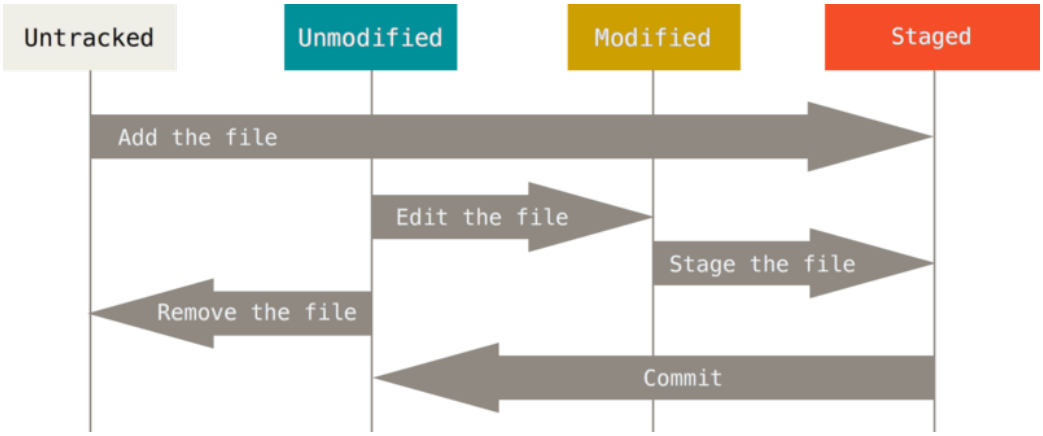

Figure 1: The `git` life-cycle.

Git work flow

First, **required reading:**
`https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F`

Basically, files (or directories) exist in one of four states of a life-cycle: untracked, staged, unmodified, or modified (see Fig. 1). The standard workflow is thus:

1. Add or modify some files.
2. Stage the new or modified files.
3. Commit the changes (moving them from the Staging Area to the "memory" of the repository).
4. Repeat.

The motto of `git` is "*commit early, commit often*". Every time you add or remove something from your project (i.e. a file or a chunk of code), you want to commit those changes. Ideally, each commit should correspond to a "logical unit", one that you are able to describe in a few words (e.g., "function to perform resampling added to analysis script"). Choosing when to commit is quite important, especially when you're debugging code: For example, if you've discovered your code has two bugs then you should commit each one of the fixes seperately, not together. That way you can undo either fix

Figure 2: Don't let this happen! (source: `http://xkcd.com/1296/`)

independently if, for example, you messed up in one of your fixes or your fix created a different bug somewhere else in your code.

Within `R-Studio` (and assuming you have your `R-Studio` project opened), looking at the `git` tab will show you a list of all the files (and directories) that have been changed, removed or added to the project since the last time you committed. Clicking on the check-boxes associated with each file will add them to the staging area. Clicking on `Diff` or `Commit` will open up a new interface (the staging area). In the top-left corner you'll see a list of the staged files. Selecting one of the files will bring its contents up in the window below which highlights the text that has been added (in green) and removed (in red). Select all the files you want to commit, then add a commit message to the top-right window, and commit.

You've now updated your local respository. Clicking on `History` (top-left) will show you all your past commits. Remember, "*commit early, commit often*".

The final thing to do (often, for back-up reasons, but not necessarily following each commit) is to `Push` your commit to `Github`. `Pull` obviously does the opposite: bringing commits that have been saved to `Github` (by others, or by you on a different machine) to your local machine.

Note: *To avoid conflicts, the safe thing to do is to always pull before you start making edits, commit and push. There are of course ways of dealing with conflicts and merging files (that will arise if two people are working on the same file, for example), but for our mostly single-user purposes it's easier to just avoid them.*

Git resources

There's a whole lot more functionality to `git`, some of which is important and very useful (such as resolving conflicts, branching and merging, and reverting back to old versions), but we're not going to get into that unless we have extra time. For more information, check out the following resources:

`https://www.codecademy.com/learn/learn-git`
`https://www.atlassian.com/git/tutorials`
`http://rogerdudler.github.io/git-guide/`
`https://git-scm.com/book/en/v2`

Coding

The reality is that for any non-trivial task, most of your time will be spent de-bugging your code, not writing it. There are two over-arching principles that will, when practiced, greatly increase your efficiency: writing *modular* code and writing *clean* code. In class we'll discuss modularity first (it relates to the *project mindset* referred to in the context of `git`) and will then talk about code-writing best practices (styles guides).

Principle of Modularity

If your experience learning to code as been anything like mine then, either when you first started coding or when you wrote homework assignments for Stats class, the code you created for a given project probably consisted of one long giant file. If you learned to use `Markdown` or `Sweave`, that code probably had some number of sections within it, just like you might write a paper or thesis chapter (Intro, Methods, Results).

That'll work fine for small (tiny) projects or homework reports, but probably not for anything the size of a paper or thesis chapter that contains some combination or subset of data, analyses (whether statistical or non-statistical modeling), and data- or results visualization.

Thus, as alluded to in Your Project folder / Respository, you should give your project a useful structure:

Your `data` folder should contain all the data you need for the project. Most of the time, this will include both the *original* unedited raw data files (regardless of their format) as well as *derived* data files (that you have used code to produce). No files should be duplicates or derived copies of each other (see Fig. 3); remember: versions will be tracked by `git`.

Your `code` folder contains your various script files. For anything but the smallest projects there should be multiple files that contain scripts for performing different tasks. For example, you might have:

`data_prep.R`	Script that pulls in the original data, pre-processes them (rearranges, correct errors, standardizes names, splits up or merges different data sets), and spits them back out in a "clean" format needed by your subsequent analyses
`my_functions.R`	Script containing the functions you have self-defined to perform your analyses
`analysis.R`	Script that sucsk in your "clean" data, performs your analyses (using built-in, self-defined and package functions), makes some quick-and-dirty figures along the way, and exports the results to `output`
`final_figs.R`	Script that sucks in the results of the analyses to produce the final figures (or tables) for your manuscript

In general, you don't want any file to become unwieldy. Thus you will likely have multiple scripts within each of the above categories (e.g., one for each of several different data set types, one for each of several different analyses or analysis steps, or one for each of several different self-defined functions) – each appropriately named (see Code style guides). If you end up with a whole lot of scripts to perform an analysis from start to finish, then you may want one additional `RunMe.R` script that sources each of the other scripts in the appropriate order.

The key utilities of separating out everything as specified above are (1) *readability* and (2) *unit-testing*. Readability means that it's easy for anyone (including you in 1 years time) to figure out where things are being pulled from and where they're going, and no individual script is overwhelming to look through. The idea behind unit-testing is to write independent tests for the smallest units of your code. For example, whenever you write a function, you also write an associated small piece of code that tests it (before the function is used in your primary analysis). That way, whenever you modify your code or function(s), it's easy to ensure that everything is still returning the correct output.

Working directory and directory access

In order to employ the principle of modularity you need to know how to navigate between your various folders and scripts. Within a project, for example, you have a choice between using the repository folder as your working

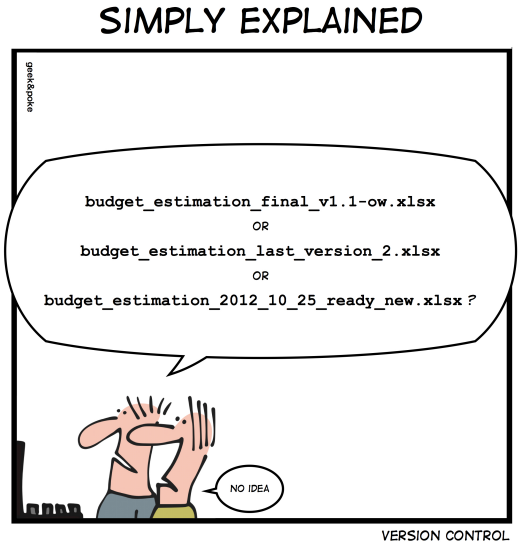

Figure 3: Never again should your data files look like this!

directory or setting your working directory to be the directory in which you have saved your scripts (e.g., your RunMe script). Your scripts will therefore need to access data and source scripts that are located in other folders. You want to set these locations as generically as possible (so that you can move your entire project to a different location without destroying the workflow, for example). You never want to set the working directory more than once, or hard-write the locations of your data or other scripts within a given script.

You can do this by specifying locations relationally using '`/.../file.R`', for example. Each repetition of '`/.../`' will move you up one level in the folder structure. For example, assuming your working directory is the `code` directory, rather than specifying the location of your data as

```
read.csv(file='C://MyDocuments/Git/MyProject/data/data.csv')
```

you should instead use

```
read.csv(file='.../data/data.csv').
```

The latter takes you up one level (out of `code` into your general `repository` folder) then into the `data` folder and to your data file. To pull in scripts, use

```
source(file='my_functions.R')
```

(assuming your working directory is the `code` directory).

Note: Macs (Unix) and Windows (Dos) use forwardslashes and backslashes differently. Macs use '`/.../.../file.R`' (forwardslashes) while Windows use '`\...\...\file.R`' (backslashes).

Code style guides

There are a lot of summarized sets of recommended best-practices for coding in general and for `R` as well. These includes aspects relating to object naming conventions (for filenames, function names, and variable names), syntax and grammar (spacing, indentation, etc.), and code structure. I won't repeat everything here, but we will go over and discuss the big ones in class. For our course, the **required reading** is:

`https://google.github.io/styleguide/Rguide.xml`

Other guides that are well worth reading for some additional suggestions and explanations are:

`https://www.r-bloggers.com/r-code-best-practices/`

and

`http://adv-r.had.co.nz/Style.html`

Within-script organization

In additional the best practices in the naming of objects and the syntax of your code, there's also an important aspect of within-code organization. Your code should consist of the following sections, each visually separated from the others:

1. Start each file with a preface that describes what it contains and how it fits into the project. You might also want to include who wrote it and when.
2. Load all required packages
3. Source required scripts (e.g., `my_functions.R`)

4. Load (or source) required data (or `data_prep.R` scripts)
5. Section(s) for major parts of your analyses
6. Export results section(s)

The last two may consist of just two sections or may have export parts immediately following an analysis. Wherever possible and appropriate, clear your workspace (`rm(list=ls())`). For example, you'd most definitely do this at the top of your `RunMe.R` script and perhaps to at the top of your `analysis.R` script (since your "cleaned" data was saved and can be reloaded, leaving all the temporary variables unnecessary), but you wouldn't put `rm(list=ls())` at the head of your `my_functions.R` script.

Thus a script file might look as follows:

```
1  ##########################################################
2  # simulateLV.R
3  # Simulates the dynamics of a predator and prey
       population according to the Lotka-Volterra model.
4  # The data produced will subsequently be used to test
       the performance of several population dynamic model-
       fitting routines.
5  ##########################################################
6  rm(list=ls()) # clear workspace
7
8  ###############################
9  # Load librairies
10 ###############################
11 library(deSolve)
12
13 ###############################
14 # Source files
15 ###############################
16 # None needed
17
18 ###############################
19 # Define model
20 ###############################
21 LVmod <- function(Time, State, Pars) {
```

```
22     with(as.list(c(State, Pars)), {
23        Ingestion    <- rIng  * Prey * Predator
24        GrowthPrey   <- rGrow * Prey * (1 - Prey/K)
25        MortPredator <- rMort * Predator
26
27        dPrey        <- GrowthPrey - Ingestion
28        dPredator    <- Ingestion * assEff - MortPredator
29
30        return(list(c(dPrey, dPredator)))
31        })
32     }
33
34  ################################
35  # Define parameters
36  ################################
37  pars <- c( rIng   = 0.2,    # /day, rate of ingestion
38             rGrow  = 1.0,    # /day, prey growth rate
39             rMort  = 0.2 ,   # /day, predator mortality
40             assEff = 0.5,    # assimilation efficiency
41             K      = 10)     # mmol/m3, carrying capacity
42
43  ################################
44  # Simulate model
45  ################################
46  yini  <- c(Prey = 1, Predator = 2)
47  times <- seq(0, 200, by = 1)
48  out   <- ode(yini, times, LVmod, pars)
49  summary(out)
50
51  ################################
52  # Plot and export data
53  ################################
54  plot(out)
55  write.csv(out, file='.../output/LV_out.csv')
56
57  ##############################################################
```

When defining a function, you should provide additional information that describes what the function does (in general terms), what inputs it takes, and what output it returns. After the function is defined it's worth adding a test case (commented out). [Note: some would separate out test cases, leaving only the function definition in its own script.] For example:

```
1  ############################################################
2  # Function to add stochastic noise to a time-series of
       population sizes.
3  ############################################################
4  # Input:
5  #   x -- a time series of population sizes (vector)
6  #   error_model -- gaussian (currently implemented model,
        default)
7  #   sd -- the standard deviation of gaussian errors (
       default=0)
8  # Returns:
9  #   Vector of length equal to the input vector of time
       series
10
11 add_error <- function(x, error_model='gaussian', sd=0){
12     error_model <- match.arg(error_model)
13     if(error_model=='gaussian'){
14         out <- x + rnorm(length(x), sd)
15     }else{
16         out <- x
17         warning(''Original time series returned'')
18     }
19     return(out)
20 }
21
22 # Test:
23 # source('simulateLV.R')
24 # new <- add.error(out, sd=1)
25 # par(mfrow=c(1,2))
26 #   plot(out)
27 #   plot(new)
```

When writing code, though shalt not...

- Copy-paste-edit (i.e. reuse and modify) the same code to repeatedly apply it to different subsets of data, etc.. Instead, you will convert your code to function(s) that can be applied to these data subsets
- Use `attach()` and `detach()` on your data. Instead, be explicit in naming and accessing data.frame columns.
- Repeatedly reset your working directory. Instead, use relational sourcing.
- Save your R workspace for later reuse. Instead, use `rm(list=ls())` wherever possible and appropriate.
- create large tables by hand. If you find yourself having to export large tables a lot, learn LaTeXand export them instead

git cheat sheet

learn more about git the simple way at *rogerdudler.github.com/git-guide/*
cheat sheet created by Nina Jaeschke of *ninagrafik.com*

create & clone

create new repository	*git init*
clone local repository	*git clone /path/to/repository*
clone remote repository	*git clone username@host:/path/to/repository*

add & remove

add changes to INDEX	*git add <filename>*
add all changes to INDEX	*git add **
remove/delete	*git rm <filename>*

commit & synchronize

commit changes	*git commit -m "Commit message"*
push changes to remote repository	*git push origin master*
connect local repository to remote repository	*git remote add origin <server>*
update local repository with remote changes	*git pull*

branches

create new branch	*git checkout -b <branch>* *e.g. git checkout -b feature_x*
switch to master branch	*git checkout master*
delete branch	*git branch -d <branch>*
push branch to remote repository	*git push origin <branch>*

merge

merge changes from another branch	*git merge <branch>*
view changes between two branches	*git diff <source_branch> <target_branch>* *e.g. git diff feature_x feature_y*

tagging

create tag	*git tag <tag> <commit ID>* *e.g. git tag 1.0.0 1b2e1d63ff*
get commit IDs	*git log*

restore

replace working copy with latest from HEAD	*git checkout -- <filename>*