Getting started with `Git`

Contents

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 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 (graphical user interfaces, e.g., `R-Studio`) rather than command-line.

Github

`Git` stores a complete copy of the project on your local machine, including all its history and 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`.[1] The former is more developed (more bells and whistles), is currently more widely used, and

---

[1]I suggest creating accounts with both `Github` and `Bitbucket`. I use them for different types of projects, e.g., public verus private, as needed.

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 are perks regarding the number of repositories you can have if you sign up using an academic email address. See https://help.github.com/en/articles/applying-for-an-educator-or-researcher-discount)

Installing and configuring Git

See the `README.md` of our very first class for installing `Git`.

After installation, 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 atom
```

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

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

For more, or if you're using Windows, 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 to interact with `Git`.[2] Instead, we're mostly 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 up the repository on `Github` and then clone it to our local master folder of projects.

---

[2]See last page for a cheat sheet.

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 see 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. We will learn to use Markdown to write and edit README files later in the course, so for now just write some minimally informative text in it without any formatting (e.g., your project title).

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. Later in the course, we'll also add the extensions for all the temporary files that LaTeXproduces when compiling.

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). We'll ignore them for now, but feel free to explore! You *could* start dragging-in directories and files into your browswer view 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 projectfolder) and go from there.

To clone the repository, click the green `Clone or download` button and copy the provided URL. There's a few ways to clone your repository to your local machine. Your preferred method depends on how you're likely 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 you just copied;

2. use a visual `Git` GUI client to clone the repo;

or, if you're primarily going to be using this repository to keep track of an R-based project using `R-Studio`:

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 haven't yet 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.

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.

Git workflow

Before proceeding, jump over and do the **required reading:**
https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F.
Then come back here.

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;

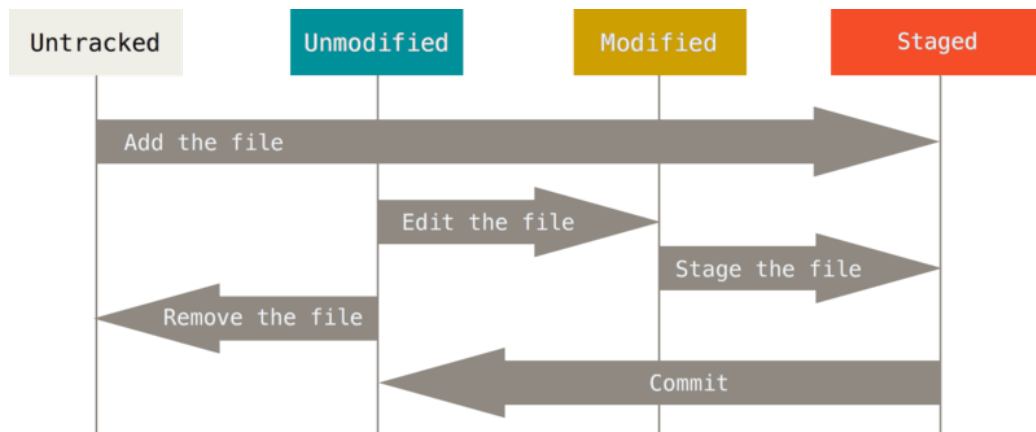3. Commit the changes (moving them from the Staging Area to the "memory" of the repository);

4. Repeat.

Figure 1: The `Git` life-cycle.

Your motto for using `Git` should be "*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., "Add function to perform resampling"). 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 independently if, for example, you messed up in one of your fixes or your fix created a different bug somewhere else in your code.[3]

`Git`-GUIs provide visual interfaces for viewing your files, staging area, and commits. Within `R-Studio` (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, stage them, add a commit message to the top-right window, and commit. You've now

---

[3]We'll talk about using "branches" to reduce the incidence of problems down the road.

updated your local respository. Clicking on `History` (top-left) will show you all your past commits.

Remember, "*commit early, commit often*" and provided concise and informative commit messages (Fig. 2).

The final thing to do (not necessarily following each commit) is to `Push` your commit to `Github`. `Pull` does the opposite: bringing commits that have been saved to `Github` (by others, or by you on a different machine) to your local machine. To reduce conflicts, *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 collaborator(s) are working on the same file, for example), but pulling first will do a lot to avoid unneccessary hassle.[4]

So again, our basic recipe is: *pull, create/edit, stage, commit, push, repeat.*

	COMMENT	DATE
	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
	ENABLED CONFIG FILE PARSING	9 HOURS AGO
	MISC BUGFIXES	5 HOURS AGO
	CODE ADDITIONS/EDITS	4 HOURS AGO
	MORE CODE	4 HOURS AGO
	HERE HAVE CODE	4 HOURS AGO
	AAAAAAAA	3 HOURS AGO
	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
	MY HANDS ARE TYPING WORDS	2 HOURS AGO
	HAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

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

---

[4]We'll learn about merging and conflict resolution later in the course.

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>*