

# Getting started with `Git` & `GitHub`

## Contents

What is version control?	2
<code>Git</code>	2
<code>GitHub</code>	2
Installing and configuring <code>Git</code>	3
Repository setup	3
R-Studio and <code>Git</code> GUIs	5
The <code>Git</code> workflow	6
Syncing with <code>GitHub</code> . . . . .	8

## 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 know 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 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 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<sup>1</sup> 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 features. We'll also interact with **Git** using GUIs (graphical user interfaces, e.g., **R-Studio**, **Sourcetree**, **GitHub Desktop**) rather than command-line.

## GitHub

**Git** stores a complete copy of the project on your local machine, including all its history and versions; no centralized server is necessary. However, if you want to collaborate with others, have multiple computers on 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

---

<sup>1</sup>Except when using the “Git-light” that comes with **GitHub Desktop**!

used, and is perhaps a little easier to work with. At the time of writing, 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

If you're using **GitHub Desktop**, then you're already done; a "light" version of **Git** is bundled into the app.

If you'd like to install the full version of **Git** to use per command-line or with a different GUI, then see the `README.md` of our very first class. After installing the full version 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 a default text editor with which to open files, for example:

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

(replacing `code` (which stands for Virtual Studio Code) for the call to your editor).

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**.<sup>2</sup> Instead,

---

<sup>2</sup>See last page for a cheat sheet.

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 or your Git GUI and then connecting it to `GitHub`, we'll create and set up the repository on `GitHub` and then clone it to our local folder of projects.

Simply login to your `GitHub` account, click on “New Repository”, and follow the instructions.<sup>3</sup> These should include options for private vs. public (the latter is preferred for this class<sup>4</sup>), initializing with a `README.md` file (which you *should* do), as well as a `.gitignore` file (which you *should* also do, choosing your planned programming environment (e.g., `R`) from the dropdown menu). You don't need to initialize with a license (MIT is a common choice, if you would like to).

The `README.md` file in the main repo folder is the first file that anyone will see when they inspect your repository. At minimum, it should give an overview of what the project is about and what the various parts of the project structure are. The initialized file will simply contain the repository's name as a header. We will learn to use Markdown to write and edit `README.md` files later in the course, so for now just leave it as is.

The `.gitignore` file contains a list of all the files that you want Git to ignore (i.e. not monitor for changes). Selecting `R` from the dropdown list will auto-populate a bunch of it for you, including files like the `.Rhistory` file that save session information and the like. You can add additional file and folder names to the `.gitignore` file as well. You might want to do this to keep your data or manuscript in a different repository but still in the same project folder (see back to our discussion of structuring projects). Later in the course, we'll add the extensions for all the temporary files that `LATEX` produces when compiling.

You should now see a new webpage – your Repository page – that shows you what's in your repository. For now it contains only your `.gitignore`

---

<sup>3</sup>When doing so, be sure you're in your own user environment and not inside our Analytical Workflows organization.

<sup>4</sup>If at all possible, please pick public (for this class) and switch to private afterwards. Otherwise, please add me as a collaborator so I can see your repo.

and `README.md` files, the latter of which has its contents displayed.<sup>5</sup> As I said earlier, there are a lot of bells-and-whistles at your fingertips here. We'll ignore them for now, but feel free to explore! You *could* start dragging-in directories and files into your browser 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 (e.g., `code`, `data`, and `results`), and go from there.<sup>6</sup>

To clone the repository, click the green **Clone** 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 and for writing manuscripts. **R-Studio** has integrated **Git** functionality, so I use a visual **Git** GUI client (e.g., **Sourcetree**, **GitHub Desktop**) for the other 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 start off by using **R-Studio**'s **Git** integration feature. (Note, however, that **R-Studio** isn't able to do some useful things which we'll cover later. Therefore you'll still benefit from using a **Git** GUI or learning the commands.)

---

<sup>5</sup>There will actually be other files in the folder when you clone it to your local computer as well, but they're hidden by default. They're files that **Git** uses for managing the repository.

<sup>6</sup>Note that empty folders will not be monitored by **Git**; they need to contain something. You can (and should) add a folder-specific `README.md` to explain its contents!

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.

After that one-time initial setup, 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. Place your repository wherever you plan to keep all your project folders (see back to our discussion of structuring projects). 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/folder (which at minimum should now be: `README.md` and `.gitignore` and a newly created `.Rproj` file).

You may now create (or move in) all your project sub-folders.

## The 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.

In a nutshell, files exist in one of four states of a Git 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 changed files (moving them from the Staging Area to the “memory” of the repository);
4. Repeat.

Your motto for using Git should be “*commit early, commit often*”. Almost 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 separately, not together. That way you can undo either

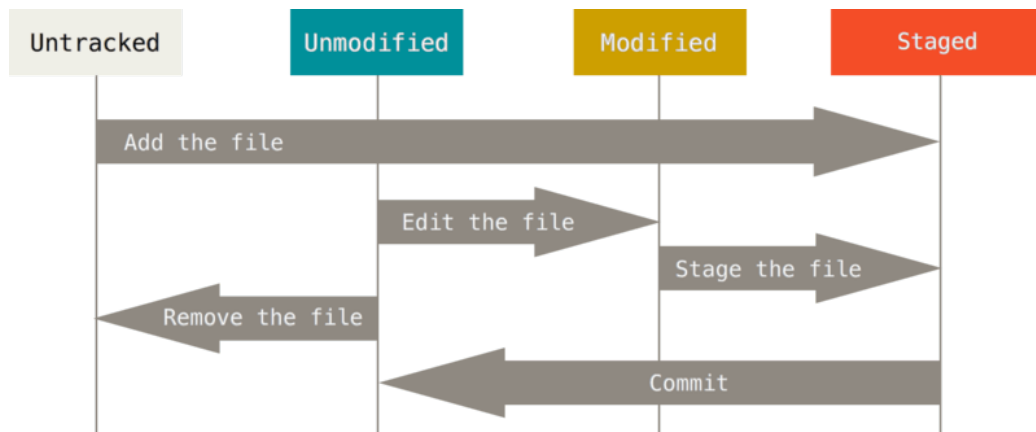


Figure 1: The `Git` life-cycle.

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.<sup>7</sup>

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

Every commit message must at minimum have a “subject line”. That subject line is often the only thing in your commit message. However, you can also write a whole lot more if you’d like, paragraphs even, by adding a blank second line between the subject line and the rest of your message. How to write good commit message subject lines is a topic unto itself (Fig. 2)!<sup>8</sup>

<sup>7</sup>We’ll talk about using “branches” to reduce the incidence of problems down the road.

<sup>8</sup>For a great post on writing commit messages, see <https://chris.beams.io/posts/>

For now, the only things we'll say are:

1. a properly-formed `Git` commit subject line should always complete the sentence: “If applied, this commit will *your subject line here...*”;
2. capitalize the first word; and
3. don't use a period at the end of the sentence.

***Commit early, commit often***  
**with concise and informative commit messages**

## Syncing with GitHub

Though not part of the `Git` workflow itself (since `Git` is only doing things locally), the final thing to do is to **Push** your commit(s) to `GitHub`. You certainly do *not* need to do this after each commit!

**Pull** does the opposite of **Push**: bringing commits that have been pushed to `GitHub` to your local machine. This will happen if you're working collaboratively or on multiple machines.<sup>9</sup> To reduce the likelihood of creating conflicts, *always* pull before you start making edits! 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 unnecessary hassle.<sup>10</sup>

***Pull, create/edit, stage, commit, push, repeat***

---

`git-commit/`.

<sup>9</sup>You can also use **Fetch** to see if new commits have been pushed to `GitHub` before actually pulling them to your machine.

<sup>10</sup>We'll learn about merging and conflict resolution later in the course.





	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
○	AAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAANDS	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/>)

# git cheat sheet

learn more about git the simple way at [rogerdudler.github.com/git-guide/](https://rogerdudler.github.com/git-guide/)  
cheat sheet created by Nina Jaeschke of [ninagrafik.com](https://ninagrafik.com)

## create & clone

<b>create new</b> repository	<i>git init</i>
<b>clone local</b> repository	<i>git clone /path/to/repository</i>
<b>clone remote</b> repository	<i>git clone username@host:/path/to/repository</i>

## add & remove

<b>add</b> changes to INDEX	<code>git add &lt;filename&gt;</code>
<b>add all</b> changes to INDEX	<code>git add *</code>
<b>remove/delete</b>	<code>git rm &lt;filename&gt;</code>

## commit & synchronize

commit changes	<code>git commit -m "Commit message"</code>
push changes to remote repository	<code>git push origin master</code>
<b>connect</b> local repository to remote repository	<code>git remote add origin &lt;server&gt;</code>
<b>update</b> local repository with remote changes	<code>git pull</code>

## branches

<b>create</b> new branch	<i>git checkout -b &lt;branch&gt;</i> e.g. <i>git checkout -b feature_x</i>
<b>switch</b> to master branch	<i>git checkout master</i>
<b>delete</b> branch	<i>git branch -d &lt;branch&gt;</i>
<b>push</b> branch to remote repository	<i>git push origin &lt;branch&gt;</i>

**merge**

<b>merge changes</b> from another branch	<code>git merge &lt;branch&gt;</code>
<b>view changes</b> between two branches	<code>git diff &lt;source_branch&gt; &lt;target_branch&gt;</code> e.g. <code>git diff feature_x feature_y</code>

## tagging

<b>create tag</b>	<code>git tag &lt;tag&gt; &lt;commit ID&gt;</code> e.g. <code>git tag 1.0.0 1b2e1d63ff</code>
<b>get commit IDs</b>	<code>git log</code>

## restore

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



### Tip

Want a simple but powerful  
git-client for your mac?  
Try Tower: [www.git-tower.com/](http://www.git-tower.com/)