An Introduction to Git for Scientists (And other regular people)

Garrick Aden-Buie*

Contents

1 Why use git? 2

2 Version control without the control 3

2.1 What about the files? 4

3 There has to be a better way 5

3.1 The Dropbox Method 5

3.2 The Google Docs Technique 6

4 The Git Way 6

4.1 Separate your work into bite-size chunks. 7

4.2 Separate your work into features. 9

4.3 Don't break things. 11

4.4 Work locally, share when you're ready. 12

4.5 What changed? 15

5 Go Deep 19

5.1 A gentle introduction 20

5.2 Great references 20

*http://garrickadenbuie.com

1 Why use git?

Git is a tool to manage and record changes to files, and it works just as well for a single person on their own personal computer as it does for hundreds of contributors collaborating together on a single project. Technically speaking, git is a (free and open source) *distributed version control and source code management software*.

But why use git? Most introductory git tutorials start by assuming that the reader is familiar with version control systems, has tried a few of them, and is now considering whether or not git is the tool they'll want to use. Usually what they're really talking about is reasons for using a Distributed Version Control System verse a Centralized Version Control System. And then they talk about git vs. Mercurial, another DVCS.

But for users who have never considered using *any* version control systems – or those who have thought about using git but have put it off because it seems too difficult – "Why use git?" is a much deeper question than "why *choose* git?"

In fact, most introductory tutorials start by walking the reader through the most basic git commands, taking for granted that the beginner has already convinced him or herself that he even *wants* to be using those basic git commands. Never mind explaining why they would use those commands to achieve what they *do* want to do, or why they would even use git at all.

This guide is intended for scientists, in particular PhD students and their collaborators, advisors and professors – and other regular people who occasionally write code to get things done. As a group, we're overworked, underpaid and busy. But we also have a much higher tolerance for learning something new to add new skills to our growing skill sets.

Doing science is rigorous and precise. Writing, editing and publishing papers is a trial-by-fire adventure in organization. Git is a powerful tool that can make your life easier:

- It makes keeping a permanent record of the history of your project possible, including each and every single change you make to your code or text.
- It makes asynchronous collaboration with others completely doable, while freeing collaborators to experiment, dabble, edit and make changes on their own time, without having to worry about breaking things or having to be connected to the internet.

- It makes your work easier and more efficient because the git workflow works best when you break your work into logical, one-item tasks.
- It makes "rolling back" changes or loading *any version* of your project state a simple, one-line command.

On the other hand, in the spirit of full disclosure, git at times seems very mysterious. It's entirely a command line tool and it requires you to have at least a fundamental understanding of what's going on behind the curtains. There are a few graphical tools available, but they still require you to understand the system.

So that's what we're going to do now. In the next section I want to show you how git works by taking a look at how most "regular" people (who have never spent an inordinate amount of time thinking about version control) approach version control. In it, we'll see that basic git isn't very different from what most people are doing now, it's just a lot more powerful and deliberate. We'll also hopefully see just how much an actual version control system can help you out.

2 Version control without the control

Let's imagine your research group lands grant funding to start a new research project. The research project involves writing code to run several simulated scenarios, after which the data generated and collected from numerous simulations will be analyzed and presented. In the end, of course, the results will have to be written up, figures and tables generated and – after a long phase of revisions and edits – submitted to a journal for publication.

Of course, three months later your group receives the reviewer's responses. One nice reviewer pointed out that your group overlooked an important metric in the simulations and your group needs to go back and run all of the simulations again – with all of the parameters and versions from the original set of simulations.

In those three months, however, your group, not about to get complacent with the research, kept working on the simulation code and is nearly ready to run these new revised simulations. After a day of looking through folders full of code files, your team finds most of the old code, but one file is missing. You'll have to stay up all night trying to remember what you did nearly half a year ago.

Let's go back to the beginning. At the start of the research your team had a group meeting and decided to divide up the tasks. One or two people went to work on

coding the simulations, one person went to work on setting up the framework for the design of experiments. Then later everybody switched to working on analysis and finally everybody jumped in on writing up the sections they were worked on.

2.1 What about the files?

Most people in these situations – people who haven't thought much about version control in general – break off into their groups and get straight to work. The simulation coding crew gets down to business, and they share their code by doing all their work in a shared folder on the server. Luckily, they sit next to each other in the grad lab, so whenever they need to edit a file that the other person is working on, they just lean over and ask how long it's going to take to finish up.

The student working on the DOE framework works from home, so whenever he needs to test out the new simulation code, he emails the simulation crew and they send him back a zip of updated files. Usually this is where they discover that things have broken, so the emails go back and forth for a while, and by the end the file name of that zip file looks like this:

```
simulation_code2013-08-18_bugfix2_final_FINAL.zip
```

Things get even worse when the analysis and writing start, especially when the advisor wants to see new results and new sections written. Emails fly back and forth, and even though *Track Changes* is turned on in the Word document, multiple revisions are being made by different people to the same version, so some kind of file name change is needed. Inevitably, one unlucky soul gets elected gate-keeper for the paper, so everybody piles their revisions on to her and she organizes them, manages conflicts, compiles the changes and then sends out the new version.

Two weeks in, the folder that she keeps all of the revisions in looks like this:

```
$ ls ~/Documents/ResearchProject/
  SimResearch_v1_Dan_comments.docx
  SimResearch_v1_Ali.docx
  SimResearch_v1.2_Dan-edits.docx
  SimResearch_v1.3_ProfZ-majoredits.docx
  SimResearch_v1.4.docx
  SimResearch_v1.4_Ali_revisions.docx
```

```
SimResearch_v1.8_rewritten_section_2.docx
SimResearch_v1.8_rewritten_section_2_ProfZ_edits.docx
SimResearch_v2_start_over_from_scratch.docx
SimResearch_v2_start_over_from_scratch_Ali_comments.docx
SimResearch_v2_start_over_from_scratch_Ali_comments_Dan_changes.docx
```

Everybody is trying their best to communicate with each other, but everybody has a slightly different idea about how to communicate.

Dan always increments the version number by .1 and tries to summarize the changes he's made to the document. But Professor Z always ends up asking for entire sections to be rewritten. Within a few days or weeks version numbers aren't descriptive enough, so Alyssa started adding descriptions to the file name, but then Ali and Dan just added their names behind those changes.

Plus, every change that is being made is sent by email, with everyone in the group CC'd. By the time Alyssa's folder looks like this, everybody on the team has an even more clogged inbox, full of messages about changes made and revisions proposed.

3 There has to be a better way

There is a better way. There are many better ways, in fact.

There are hundreds of online collaboration tools available that would make the lives of everyone on the research team better. The two most common ways are the Dropbox Method or the Google Docs Technique.

3.1 The Dropbox Method

In the Dropbox method, there is one shared folder for the team[1]. All of the files in that folder automatically sync for everyone in the team. This solves two problems:

- Everyone on the team always has up-to-date files.
- Everyone on the team always has access to the files, online and offline.

[1] I said *Dropbox method*, but there are plenty of other services that do this same basic procedure.

It also *doesn't solve the problem of version control.*

Two people can work on the same file in Dropbox, but this will generate conflicted files.

Dropbox caches 30 days of file history, so you can jump back to any version of the file in the last 30 days, but it doesn't keep track of *what you did to the files*, or *why you did what you did to the files*. Your group will still have to track those changes somewhere else, and most likely they'll do this by adding version numbers and descriptions to file names and sending emails about changes.

3.2 The Google Docs Technique

The Google Docs technique is *real-time* collaboration. Files are hosted online, in the cloud, and anybody can open the file and start editing. Changes are synced in real-time for anybody else using the file, thus resolving the one-person-at-a-time problem.

It doesn't completely resolve the tracking changes problem. You can see who changed what and when, but, after several rounds of editing, the document will be a mess of changes. Edits made to the beginning and end of the document may be part of the same "logical task", but good luck trying to figure out who did what and why.

The other issue is that this style of collaboration only works well on documents that you would *want* to edit at the same time with someone else. Typically, that's limited to text documents, where writing content together is important. So this method generally doesn't help you keep track of coding changes.

4 The Git Way

The Git Way is actually very similar to the natural flow of the research group…*only more deliberate*. I'll elaborate on how git can be inserted into the research team's flow and introduce some git terminology as we go.

Some ground rules first. There are a million different resources for learning git. There are plenty of great resources to learn the detailed ins and outs of git – to wit, the Pro Git book.

Git is a powerful and very flexible tool. Almost too flexible. Git works for nearly *any* workflow, for any size project, from keeping track of just one file on your own system to colossally huge projects like the Linux kernel.

As a result, there are plenty of resources that tell you *how* to do things with git that assume you know both *what* you want to do and *why*. My goal is that after reading this you'll have an idea of how git can fit into your workflow. You'll have a vague notion of git concepts and how they can do a better job of doing the things you're already doing. But it's really just a glimpse.

Then I'll point you in the direction of good resources that do a much better job of covering the details. You can take it from there.

Let's go back to your research group and see how git can work for you.

There are two ways to get started in git. If someone on your team has already created the repository and has hosted it online, you can create a clone of the git repository on your local machine by running `git clone <url>`. The clone command creates a folder and puts all of the data in the repository in that folder. For example, to clone the git repository for this project you would run:

```
$ git clone http://bitbucket.org/gadenbuie/intro-to-git-for-scientists
```

On the other hand, if you're creating a repository for the first time – even if the code already exists – you initialize a git repository by going to the project's directory and typing:

```
$ git init
```

This creates a `.git` directory in the project folder, but nothing in your folder is tracked yet. In the next section we'll see how to add files to the repository and track the changes made to them.

4.1 Separate your work into bite-size chunks.

There is nothing worse in life than a vaguely defined, overly general task item, looming at the top of your task list. Every time you look at it, it might as well say: *Solve world peace*. You'll never get it done in your lifetime.

The best way to use git is to, instead, break down your task list into single item bullet points:

- Write a new random number generator function.
- Create a results object class.
- Write a function to save results as a .csv file.

Each of these tasks is a logical entity in the history of your project. They also don't overlap much. As you work through your task list, every time you finish writing the code for a list item above – once you've *tested the code* to make sure it does what you want it to do – you stop and tell git to save the state of your files.

Let's say you were working on the file `simulator.py`. When you finish writing the random number generator, you tell git to save your work:

```
$ git add simulator.py
$ git commit -m 'Rewrote random number generator function'
```

`git add` takes a snapshot of `simulator.py` and **add**s it to your *staging area* in the state that it's in at that moment. Then, `git commit` **commit**s those changes to the "permanent" record, with the comment `Rewrote random number generator function` attached.

You can do more with both commands. You can selectively add files to the staging area with `git add` until you have a snapshot of all the files you want in your next commit. When files are added to the staging area, a snapshot is taken right then. If you change a file later and you want those changes included in the commit, you have to add them again.

If you run `git commit` without any arguments, git opens an editor for you to write your commit message. Take advantage of this. In general, it's good to think of it like a small email to yourself. Write out a descriptive subject line, skip a line, and include a more detailed description of the changes in the commit. For example:

```
Rewrote random number generator function

* Removed middle square method from random number generator
  function in simulator.py
* Replaced with linear congruential generator method
    * From "Numerical Recipes: The Art of Scientific Computing",
      Press et al., 2007.
```

When you save the file and close the editor, the commit is finished and your message is stored. Later, at the weekly team meeting when you're trying to remember what you did last week, you can run `git log` and see a descriptive, logical, sequential list of everything you've done[2]. It's almost magic.

> **Pro tip:** Commit early and commit often.

4.2 Separate your work into features.

Whenever you start to work on something *new* – like a new idea, a new feature, a brilliant new idea that came to you in the shower, or that off-the-wall idea the team cooked up in the last meeting – you'll want to keep the feature separate from the main code until it's ready for prime time. Then, when you're ready you'll fold in the new feature.

With git, you get this done by starting a new **branch**. In git, a branch has two defining characteristics:

1. A branch has a **parent**. In other words, a branch starts from somewhere, that somewhere being a certain state of the codebase.
2. A branch contains a record of the changes made on top of that starting point.

In git, branching is *cheap*, meaning that it's easy to create a branch. They don't take up any extra space and there is no good reason for *not* using them.

Anytime you're about to make a group of changes to your code – like before you start on that task list above – start a new branch with:

```
$ git branch awesome_new_feature
$ git checkout awesome_new_feature
```

The first command, `git branch`, marks the starting point for the new branch. Basically, it says to create a new branch, named `awesome_new_feature`, that starts with the current state of the code.

[2]This is also really good for when you want to take a look at someone else's code to see what *they've* done before you allow those changes into your project.

The second command makes it easy to switch between branches, or even individual commits. The checkout command, `git checkout`, tells git to change all the files in the project directory that are tracked by git[3] to the state they were in at whatever label you use. If you've just started a new branch, nothing changes because you haven't made any commits to *awesome_new_feature*, but `git checkout awesome_new_feature` tells git that any new commits you make are commits to your awesome feature branch.

But let's say you work on *awesome_new_feature* for a while, adding one commit for each of the items on your `simulator.py` task list. Somebody emails you and asks you about code that was part of the old random number generator function, so you want to get back to the code as it was before you started making your awesome changes.

To do this you can use:

```
$ git checkout master
```

In git, there is generally one branch called master. This where most developers decided to keep the good code; the features that have made it past testing phases are all here. It's recommended (see the next section) to try to keep the master branch full of clean code only.

The checkout command changes all the files to match the state they were in at whatever pointer you give it, so when you use `git checkout master` all of the tracked files in your directory magically change back to the state they're in at the `master` branch. To get back to your work in the feature branch, you just check it out again: `git checkout awesome_new_feature`.

> **Pro tip:** If you ever get lost, simply run
>
> ```
> $ git status
> ```
>
> It tells you
>
> - what branch you're on,
> - what files are in your staging area, and
> - what files have been changed since the last commit (but not added to the staging area).

[3] As in, all the files you've already committed.

4.3 Don't break things.

When you've worked out all the kinks of your new feature and are ready to fold the new code from the feature branch into the master branch, you do this:

```
$ git checkout master
$ git merge awesome_new_feature
```

This switches your code back to the state of the master branch and then merges the new feature branch into that code. As long as there haven't been any changes to the master branch, all of the commits from the *awesome_new_feature* branch will be **merge**d into the master branch.

What if things *have changed* in the master branch? Maybe while you were working on your awesome new feature, you found a bug in the code. But it wasn't related to your feature, so you checked out the master, made the change in that branch and then committed to the master. But you didn't realize that you later change that same line in your feature branch. Now you have a conflict.

This is the don't break things phase of the merge. Git tells you that there is a conflict between the two branches (and it's really good at merging, so if it says there's a conflict then there is certainly a conflict). Then it pauses the merge and tells you to fix the conflict.

Your job is to then open the conflicted file, find the part that git has marked as a conflict, enter the right code and save the file. To tell git that you've resolved the conflict and you're sure that the new code works, you `git add` the previously conflicted (but now resolved) file back to the staging area and then `git commit` the changes to finish up the merge.

```
$ git add conflicted_file.r
$ git commit
```

> **Pro tip:** Git doesn't want you to lose files, so it won't let you switch branches without making sure that you're working directory is clean and all the changes you've made have been committed. There are two (or more) ways around this. If you've made changes to a file but you want to switch branches you can roll back your changes by running:

```
$ git checkout -- nevermind.py
```

Or stash your changes for later with:

```
$ git stash
```

This keeps a copy of your current state in a stash list, and rolls back your working directory to your last commit. Then to get back to where you were, you can use:

```
$ git checkout branch_name  #checkout a different branch
# mess around here for a while
# when you're done:

$ git checkout awesome_new_feature
$ git stash apply
```

4.4 Work locally, share when you're ready.

All of the work I've discussed happens locally. Git makes you make the choice about what files to share, what files to change and what changes to accept into your files. This is where the distributed part of the **D**VCS comes into play.

All of your code is kept in your local project directory, or **repository**, and it's updated only when you want it to be. Your colleagues may have an exact clone of your repository, or they may have made their own changes, or they may even have outdated code.

But git doesn't care. That's right, *no repository is special.*

While it's customary for developers to have one distinguished repository with the latest production code and nearly-ready feature branches, no repository gets special treatment unless you decide it does.

What I mean by this is that git doesn't force you to stay up to date with anybody else. It doesn't force you to make changes to your code that you don't want.

This means two things. First, you have to tell git explicitly to share your code when you want to share it – for example if you want it to be available to a colleague or

you want to put your changes into the production repository. Second, you have to tell git explicitly to go fetch the changes that you want and then you have to tell it to apply any changes it has fetched to your code[4].

This might sound like a lot of work, but it's worth it. It means that you'll never have your work undone by somebody else saving an older version of a file on top of your new version. You'll never have someone else's changes forced on you without your deciding you want to deal with them. And, it reminds you to test your code and make sure it works before you make your changes public. This way you never have to share anything until you're ready.

Just like the research team's email method, you don't want to email code you haven't tested to your whole research group. And just like in the email method, you don't have to update your code with that new file from the master's student who tends to create more bugs than he fixes.

Let's say you've already set up git so that it knows where the "central" team repository is. You've also connected with Dan and Alyssa's personal repositories because you work closely with them. Lastly, you have your own online repository hosted on GitHub or BitBucket.

To fetch code from Alyssa's remote repository, you call

```
$ git fetch alyssa
```

This fetches all the commits, branches, tracked files from Alyssa's repository into your local repository. But it *doesn't apply any of the changes or affect your repository at all.* All it does is grabs a copy of her repository so that it's available for you to browse and merge into your own at your leisure.

Before you merge Alyssa's commits into your code, you might want to take a look at her files first. To see the remote branches you are tracking[5], run

```
$ git branch -r
  origin/master
  origin/awesome_new_feature
```

[4]Well, you can do this all in one step, but it's useful to do it separately to remind yourself that you're in control of your own destiny.

[5]To initialize tracking of remote repositories you use:
`git remote add <remote_name> <url_to_remote.git>`

```
  alyssa/master
  alyssa/experimental
  dan/master
  team/master
```

The output from this command shows that you are tracking 4 remote repositories named *alyssa*, *dan*, *team* and *origin*. The last one, *origin*, is simply the name that git gives to the *origin*al remote repository. This is generally the remote you cloned from or first pushed to.

Alyssa told you to check out her experimental branch because it might be useful to your awesome new feature. In order to checkout her branch, you have to create your own branch in your local repository based on the remote branch. You do this by using[6]:

```
$ git checkout -b alyssa-experiment alyssa/experimental
  From git://bitbucket.org/alyssa/simulator.git
    02e5f...91820d  alyssa-experiment       -> alyssa/experimental
$ git checkout alyssa-experiment
$ git merge github/alyssa-experiment
```

This creates a new branch in your local repository, that knows how to communicate with Alyssa's `experimental` branch. Then you checkout your local branch and merge Alyssa's branch into your new branch called `alyssa-experiment`.

You take a look at her changes and decide they look good and you want to merge them into your awesome new feature. So you go back to your feature branch and then merge your local copy of Alyssa's experimental branch into the feature branch.

```
$ git checkout awesome_new_feature
$ git merge alyssa-experiment
```

To update branches in a remote repository, you use:

[6]The general format is:
`git checkout -b <new_local_branch> <remote_name>/<remote_branch>`

```
$ git push <remote> <local-branch-name>:<remote-branch-name>
```

The most common situation will be for you to push your changes to to your own, online repository. To do this you can use:

```
$ git push origin awesome_new_feature:awesome_new_feature
```

If the local and remote branches have the same name, then you can shorten this command to something a lot easier:

```
$ git push origin awesome_new_feature
```

> **Pro tip:** It's fairly normal to have a "central" repository that you update regularly, for example a repository on GitHub or BitBucket where you host your project or store your code. Thus, the most common way taught to push changes to or pull changes from that repository is to use:
>
> ```
> $ git pull
> $ git push
> ```
>
> When you're only moving commits back and forth between your local **master** branch and **origin/master**, then this is acceptable shorthand, but only because you already know exactly what has changed and how (because you're the one who changed it).
>
> In all other cases, you should `git fetch` and then `git merge`. Actually, even if you *are* the only one making changes, it's still usefull to `git fetch` and then `git merge`. For more details, here's an excellent discussion on why this is a good habit.
>
> The short version is that `git fetch` pulls in all of the commits from the remote repository, but gives you plenty of time to think about what you're doing before you start merging changes into your own code.

4.5 What changed?

Git provides a number of really good tools for finding out and keeping track of what has changed in the code.

4.5.1 Version numbers

Your research team has been trying to do their best by changing file names to have version numbers and little messages at the end. We've already seen how `git commit` messages can be used to make those file names seem paleolithic. But there is another feature to git that makes version numbers a matter of convenience only.

Each and every commit gets an absolutely unique identifier that is based on information about the commit and that will be the same for *anybody* with the same version of the source tree.

The commit identifier, called an *object name* looks something like this:

```
1e28346f663d609932c981e0aa3aacbe541e096e
```

But this is a little long-winded, so you can use the first several characters of the commit, as long as it is unique to your repository. For example, you can just call that commit above `1e2834` because no other commit in your repository starts with those characters.

4.5.2 Tags

A very useful feature for scientists would be to specifically mark versions of code that were used to run simulations that are later included in published results. In git, these marks are called **tags**, and they would have saved you a long night of work when that reviewer asked you to go back and modify the code used in that paper your team submitted.

When you're about to run code to generate results, mark the current commit by running

```
$ git tag -a run1 -m 'First run for results'
```

The `-a` flag specifies the annotation for the tag, in this case `run1`, and the `-m` flag shows the tag message (just like a commit). You can add longer messages by omitting the `-m` flag, as in `git tag -a run1`.

You can also go back and tag commits after you've moved past them. Suppose your history looks like this:

```
$ git log --pretty=oneline
  15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
  a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
  0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
  6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
  ...
```

You ran results when you merged the branch *experiment*, so you want to go back and tag that commit. All you need to do is specify part of the commit's object name at the end of the tag command:

```
$ git tag -a run1 -m 'First run' 6d52a27
```

Then, three months later, when the reviewer comments come back, you can easily go back to the code that was used in that paper by simply checking out the tag:

```
$ git checkout run1
```

> **Pro tip**: Tags aren't automatically pushed to remote repositories. To share tags, you need to run:
>
> ```
> $ git push <remote-name> [tagname]
>
> # For example:
> $ git push origin run1
> ```

4.5.3 `git diff`

There are a number of ways to see what's changed in your repository, either between your working directory and the last commit, or between your working directory and currently staged files, or between any two commits.

Here is a summary of the commands you'll want to use:

```
$ git log
$ git status
$ git diff
$ git diff --staged
$ git diff 6d52a27..1502795 --
```

`git log` Shows you your commit history in the current branch.

`git status` Shows you your current branch, the currently staged files, and all the files that have been changed or are not being tracked in your working directory.

`git diff` Shows you the changes that have been made to the working directory that haven't been staged yet.

`git diff --staged` Shows you the changes that have been made between the previous commit and the files currently staged to go into your next commit.

`git diff 6d52a27..1502795 --` Shows you the changes between the two commits in your history. In general: `git diff <early_commit>..<late_commit> --`

4.5.4 Undoing Things

Sometimes you'll commit a couple of changes and suddenly realized you forgot a single semi-colon or period somewhere. Rather than enter a whole new commit with an `'Oops'` message, you can just do something like this:

```
$ git commit -m 'intial commit'
$ git add forgotten_file.py
$ git commit --amend
```

To remove a file from the staging area that you're not ready to commit yet, you can just do this:

```
$ git add too_early.py
$ git status
# On branch master
```

```
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   too_early.py

$ git reset HEAD too_early.py
```

You can see that git gives you a little reminder message when you run `git status` about how to unstage a file.

Let's say you make some changes to a file, but you realize later that you don't want to save those changes. In fact, you want to throw away your changes and just get back to the last committed version of the file. To do this you just use:

```
$ git checkout -- simulator.py
```

This copies over your changed `simulator.py` file with the version of the file that was in the last commit. All your changes are gone. (So be careful with this one. And don't run with scissors.)

5 Go Deep

I'm hoping that by now you can see how git can work for you and your research group. If you've gotten this far and think you're ready to start using git to manage your source code, then there are a number of great resources on the web to help you get started.

I've ignored a lot of what happens under the hood with git. But it's fascinating and it's also very much worth having some understanding of what's going on. Geeking out on the git structure will certainly help when you realize you want to do something but don't know the exact command.

One particularly interesting aspect of git is that it's a directed acyclic graph. This means that every commit knows its parent commit (but not the other way around).

For a long detailed read on the git internals, I suggest Charles Duan's Understanding Git Conceptually.

But first, why not practice the basics?

5.1 A gentle introduction

To try the basic git commands online, I suggest code school's Try Git Course. It's an excellent interactive tutorial on the basic commands I discussed above.

Then, I recommend reading Mark Longair's short introduction to git. It's available as a PDF from his website. Just skip straight to Section 3, *Basic Principles*. He includes more details than I did here, and he does a great job of explaining all the basics of using git that you'll need to know.

Lastly, for an interactive tutorial that will teach you how to use branches, merge and rebase effectively, with an interactive shell and a matching animation of your source tree, check out Learn Git Branching.

> **Pro tip:** For users of R, I highly recommend using RStudio as your IDE. It allows you to easily enable source control with git from within RStudio.
>
> The only caveat is that you can only commit, push/pull and checkout branches from within RStudio. To do the other (not that much more complicated) tasks with git, you'll need to open a command window – for example to create a new branch or add a remote branch. Luckily, there's an easy-to-reach "Shell..." menu item under the "More" button in the "Git" tab.
>
> Here's a great tutorial to get you started with git and RStudio.

5.2 Great references

The absolutely best git reference book is Pro Git by Scott Chacon, which available in its entirety online.

Also available at the git webpage is the git reference documentation, which is conveniently separated by git function and task.

Another excellent reference tool is the Visual Git Cheat Sheet.

John McDonnell's Git for Scientists is a good, but dense, introduction to git. It does feature a number of very helpful visualizations of a git source tree.

Lastly, when in doubt, if none of the above have the answer your looking for, StackOverflow is the place to go.

Oh yeah, and go here to download and install git.

–
Garrick Aden-Buie
August, 2013
http://garrickadenbuie.com