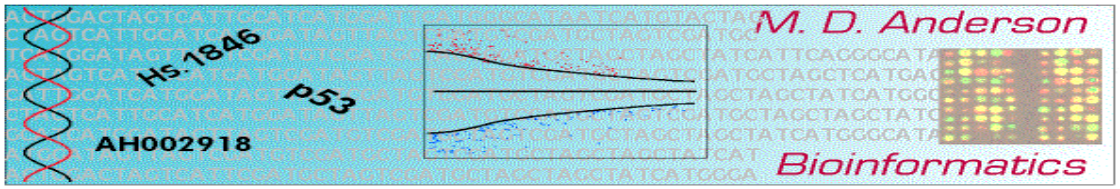
Using Git for Version Control (Part 1)

Keith A. Baggerly
Bioinformatics and Computational Biology
UT M. D. Anderson Cancer Center
kabagg@mdanderson.org

SISBID, July 21, 2015



Version Control and Sharing

Version Control tends to be one of the tools people start using later, not earlier. Let's change that.

Today, we'll

- start using git to put our projects under version control
- edit and update our working copies
- push our projects to GitHub so others can see them
- contribute to packages others have written
- include suggestions from others in our own packages

The Basic Problem

Who here has files on their computer with names like

report.doc
reportV2.doc
reportV3.doc
reportV4Final.doc
reportV5FinalKB.doc

Who here has never had to debug a script you're sure was working before?

Who here has worked on collaborative projects without confusion?

Git

These problems are addressed most directly by *version control*, and the use of a version control system (VCS).

The most widely used VCS today is *git*.

Git was introduced by Linus Torvalds (of Linux fame) to help manage large programming projects with contributions from lots of programmers (e.g., the Linux kernel).

Git lets several folks work on a project at once, and keeps track of *what* changes are made by *who* and *when*.

Git can't do everything, but in cases where changes conflict it can draw human attention to them.

GitHub

While there's definitely value in using a VCS for your solo projects, such systems really shine when you work collaboratively.

GitHub is the de facto standard location for sharing projects (repositories, or "repos") in development using git.

Putting your work on GitHub lets others see what you've done, poke at it, and make comments and/or suggestions you can choose to incorporate.

Why Aren't More People Using Git?

Lack of awareness

Inertial barriers (It looks hard!)

Occasionally alien syntax

Command-line interface

Fear/Uncertainty/Doubt?

Installing Git

We're not going to spend time on this here since there are very good instruction sets out there and we can't really do it better. So, here:

```
https://git-scm.com/book/en/v2/
Getting-Started-Installing-Git
```

Once git is installed, there are two things to set up right away:

```
git config --global user.name "Keith Baggerly"
git config --global user.email kabaggerly@mdander
```

Please check that your version is fairly recent; I'm working with 2.3.5

Now let's tour the basics!

Our Toy Example

To illustrate the very basics of git, we're going to consider the simplest case possible - a folder ("ToyFolder") containing a single text file ("toy.txt") with (initially) a single line of text

```
ToyFolder kabaggerly$ ls
toy.txt
ToyFolder kabaggerly$ more toy.txt
Once upon a time
```

Let's say we want to put this folder (and at least some of its contents) under version control.

git init

We initialize our first repository (repo) using “git init”.

```
$ git init
Initialized empty Git repository in /Users/
  kabaggerly/Repro/TestGit/ToyFolder/.git/
$ ls -a
. .. .git toy.txt
$ git status
On branch master
Initial commit
Untracked files:
  (use "git add <file>..." to include in what wil
toy.txt
nothing added to commit but untracked files prese
```

git add

While we've set up a repo, initially it's an empty shell - git is waiting for us to tell it explicitly just what we want to track.

We specify the files to track using "git add"

```
$ git add toy.txt
$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
new file:   toy.txt
```

The file has now been "staged", but not committed.

Tweaking things: Add One Word

```
Once upon a time
It
$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
new file:   toy.txt
Changes not staged for commit:
  (use "git add <file>..." to update what will be
  (use "git checkout -- <file>..." to discard cha
modified:   toy.txt
```

The changes aren't in the staged version

git add and git commit

```
$ git add toy.txt
$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
new file:   toy.txt
$ git commit -m "My first commit"
[master (root-commit) 1ce43c0] My first commit
 1 file changed, 2 insertions(+)
 create mode 100644 toy.txt
$ git status
On branch master
nothing to commit, working directory clean
```

git log and the SHA

```
$ git log
commit 1ce43c05fe699619cb71c3675d3521de1b273482
Author: Baggerly,Keith A <kabagg@mdanderson.org>
Date:   Sun Jul 12 18:37:54 2015 -0500

    My first commit
```

Each commit (a “snapshot” of the working directory) is labeled with a secure hash (SHA) assembled from a checksum of the files stored, the time, message, and so on.

This is a fingerprint - any data tweak changes the hash.

git tag

Once we've added more commits, we can refer to any snapshot by referencing its SHA, but that tends to be bulky.

We can label a given commit with a more readily memorable "tag", and use that instead.

```
$ git tag v0.01 1ce43c
$ git show v0.01
commit 1ce43c05fe699619cb71c3675d3521de1b273482
Author: Baggerly,Keith A <kabagg@mdanderson.org>
Date:   Sun Jul 12 18:37:54 2015 -0500
    My first commit
(other stuff)
```

The unique SHA prefix was enough to id it

git tag -a

We can also add “annotated” tags, which record our name, email, and a separate message.

```
$ git tag -a v0.02 -m "my version 0.02"
$ git show v0.02
tag v0.02
Tagger: Baggerly,Keith A <kabagg@mdanderson.org>
Date:   Sun Jul 12 18:54:38 2015 -0500
my version 0.02
commit 1ce43c05fe699619cb71c3675d3521de1b273482
Author: Baggerly,Keith A <kabagg@mdanderson.org>
Date:   Sun Jul 12 18:37:54 2015 -0500
    My first commit
```

Tweak, add, commit, repeat

v0.03 “was”, v0.04 “a”, v0.05 “dark”

```
$ git show -p
commit a8b333b63d9e1f31ff4a69ee16c81c234365367e
Author: Baggerly,Keith A <kabagg@mdanderson.org>
Date:   Sun Jul 12 19:12:48 2015 -0500

    added dark

...
@@ -1,2 +1,2 @@
 Once upon a time
-It was a
+It was a dark
$ git show -p -2
```

Branching out

Up to this point, we've been making changes linearly.

On occasion, however, it's useful to split things up. For example, one might have the central "master" code which is always production ready (it runs), and concentrate fragile development code in "branches" off to the side which can be "merged" back into the master version when stable gains have been locked in (this can also be associated with version increments).

git branch, git checkout

git branch creates a branch or lists those that exist, git checkout selects the active branch, git branch -b does both.

```
$ git branch
* master
$ git branch devel
$ git branch
  devel
* master
$ git checkout devel
Switched to branch 'devel'
$ git branch
* devel
  master
```

Growing the branch

```
$ git add toy.txt
$ git commit -m "added and"
[devel f52b11d] added and
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git tag v0.05.1 f52b11
```

Since I was working in the development branch, I incremented the minor version number.

Now let's say things are stable, and we want to pull the developed changes into the master line for deployment.

git merge (part 1)

```
$ git checkout master
Switched to branch 'master'
$ git merge devel -m "my first merge" --no-ff
Merge made by the 'recursive' strategy.
 toy.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

git merge (part 2)

```
$ git show -p -2
commit 27547f643495f1c2ef1691e998e091b684a13b7d
Merge: a8b333b f52b11d
Author: Baggerly,Keith A <kabagg@mdanderson.org>
Date:   Sun Jul 12 19:34:43 2015 -0500

    my first merge

commit f52b11d0f3cca99f038d990ce6c5bf1e13f617a7
Author: Baggerly,Keith A <kabagg@mdanderson.org>
Date:   Sun Jul 12 19:29:18 2015 -0500

    added and
$ git tag v0.06 27547f
```

Since the merge changed master, we're at v0.06

Now for Something Tricky (1)

Now say we tweak both the devel and master branches before we try merging. What happens?

```
$ git checkout devel
Switched to branch 'devel'
$ open toy.txt
$ git add toy.txt
$ git commit -m "added sturmy"
[devel d6c5d23] added sturmy
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git tag v0.06.1 d6c5d2
```

Now for Something Tricky (2)

Repeat the devel process with master

```
$ git checkout master
Switched to branch 'master'
$ open toy.txt
$ git add toy.txt
$ git commit -m "added stormy"
[master cc104d9] added stormy
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git tag v0.07 cc104d
```

now try a merge...

Conflict!

```
git merge devel -m "try second merge"
Auto-merging toy.txt
CONFLICT (content): Merge conflict in toy.txt
Automatic merge failed; fix conflicts and
  then commit the result.
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
Unmerged paths:
  (use "git add <file>..." to mark resolution)
both modified:   toy.txt
no changes added to commit (use "git add" and/or
  "git commit -a")
```

What our File Is Now

```
$ more toy.txt
Once upon a time
<<<<<<< HEAD
It was a dark and stormy
=======
It was a dark and sturmy
>>>>>>> devel
```

Since both branches modified the same file in different ways since the point at which they branched (v0.06), git doesn't know what to do. So it highlights the region of conflict for you to edit before continuing the merge.

Edit, git add, git commit

```
$ open toy.txt
$ git add toy.txt
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)
nothing to commit, working directory clean
$ git commit -m "new message"
[master a4b52aa] new message
$ git status
On branch master
nothing to commit, working directory clean
```

Pruning Branches When Complete

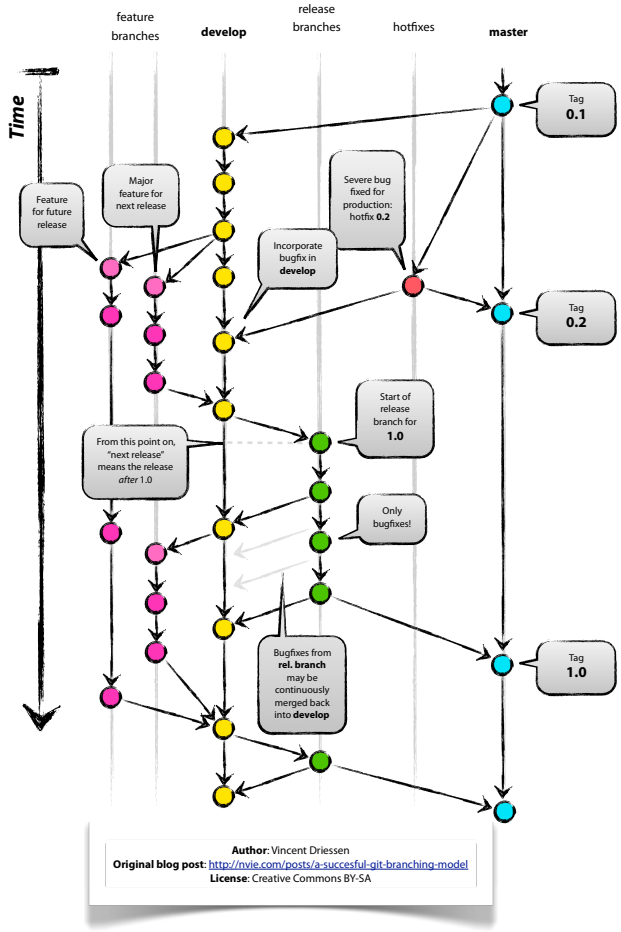
```
$ git branch
  devel
* master
$ git branch -d devel
Deleted branch devel (was d6c5d23).
```

and we're back to a clean line!

More generally, projects can involve multiple branches allowing for concurrent development by multiple users with occasional consolidation.

Let's take a look at some pro work...

If you want to know more



An Excellent Discussion

More step by step Pictures

In this case, the authors of *Pro Git* have assembled a really nice set of step by step pictures showing how branching, shifting focus, and merging can proceed in a development environment.

The book is online, so if we go to Chapter 3 `https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell` and work from Figure 3 onwards, the pictures in 3.1 through 3.4 are fun to play with (note: typo in Fig 3.9, “Divergent History”).

To support the authors, you could buy the book also.

Git and the Command Line

Git is, at its heart, a command line utility, and some of its functionality can really be accessed most directly through a shell interface. Thus, we'll use shell commands as needed (e.g. above).

That said, for most of the stuff you do day to day, graphical interfaces can help clarify much of the structure, and make it easier to remember what needs to be done.

For every pictorial example below, there is an equivalent set of command line functionality.

Git With Less Pain

Fortunately for us, Rstudio is set to interface with git directly, and it can reduce these barriers to use.

Let's explore this using the toy R package we built yesterday.

We begin by going to
Tools/Version Control/Project Setup

and setting the version control system from "None" to "Git"

It will ask you to "Confirm New Git Repository" and "Do you want to initialize a new git repository for this project?" We do.

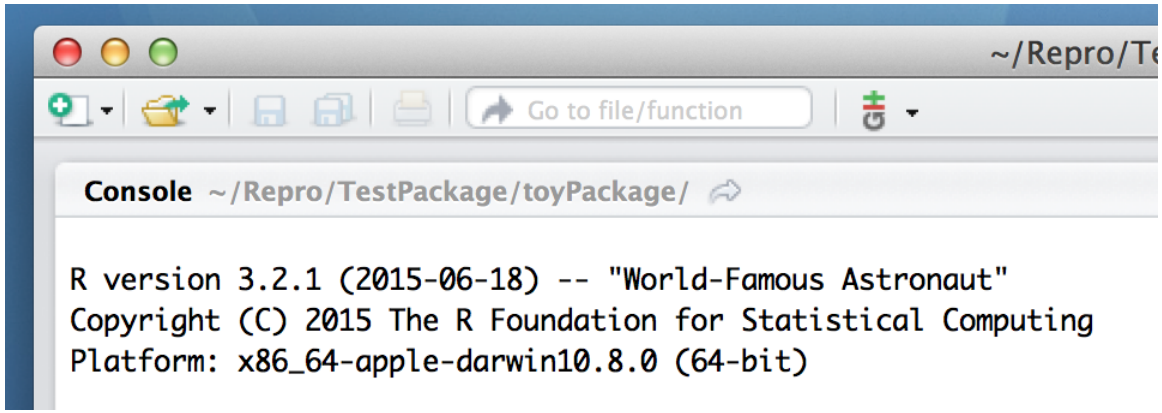
It will ask to restart Rstudio. This is fine.

What Just Happened?

Behind the scenes, Rstudio just “initialized” a git repository for this project.

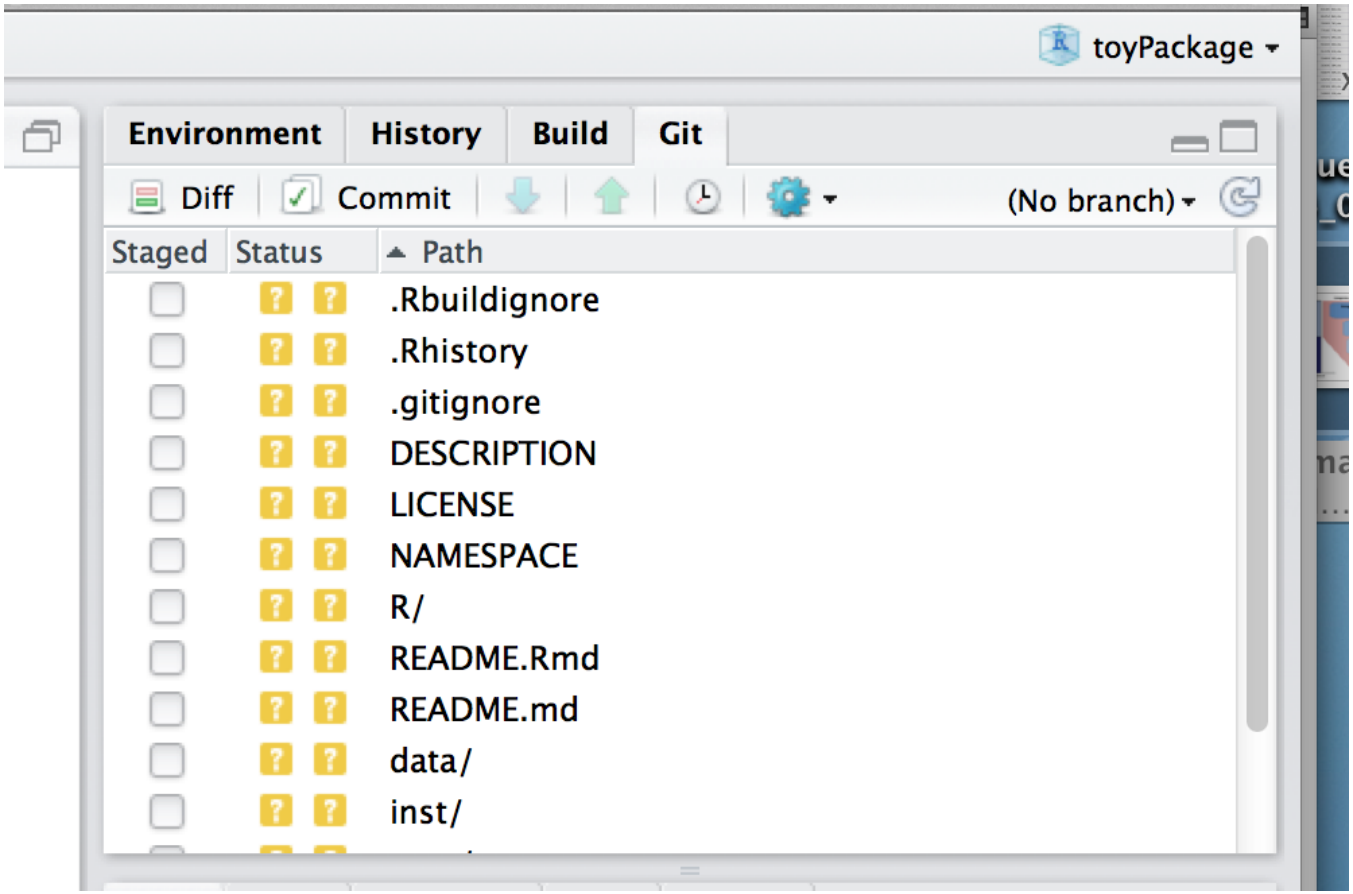
This visibly changes two things:

1. A tiny colorful Git icon appears atop the console panel



What Just Happened? (2)

2. The upper left Rstudio panel now has a “Git” pane



What Changed Less Visibly

Rstudio/Git created and populated the folder ".git" in the main project directory.

This is where the actual backups are stored, and *it should not be edited manually*. Rather, it should be edited only through Rstudio/Git itself. The .git folder is not shown in the "Git" pane, or in the "Files" pane below.

Rstudio edited the file ".gitignore" to include ".Rproj.user".

Interestingly, it might not've needed to *create* this file, because creating a vignette may have already created it with an entry of "inst/doc".

Why are my Files “??”

When the package is first put under version control, R and Git don’t know what to do with files and folders already in the project - you may not want *all* of them under version control, so it’s looking for input from you in this regard.

Once we’ve clarified which files are to be tracked, we’ll mostly just see one entry per row: Rstudio shows “staged” changes on the left, and “unstaged” changes on the right.

Since R doesn’t even know which files are to be included (staged or unstaged) at all, there are two symbols under “Status”, both of which are question marks.

Stage and Commit

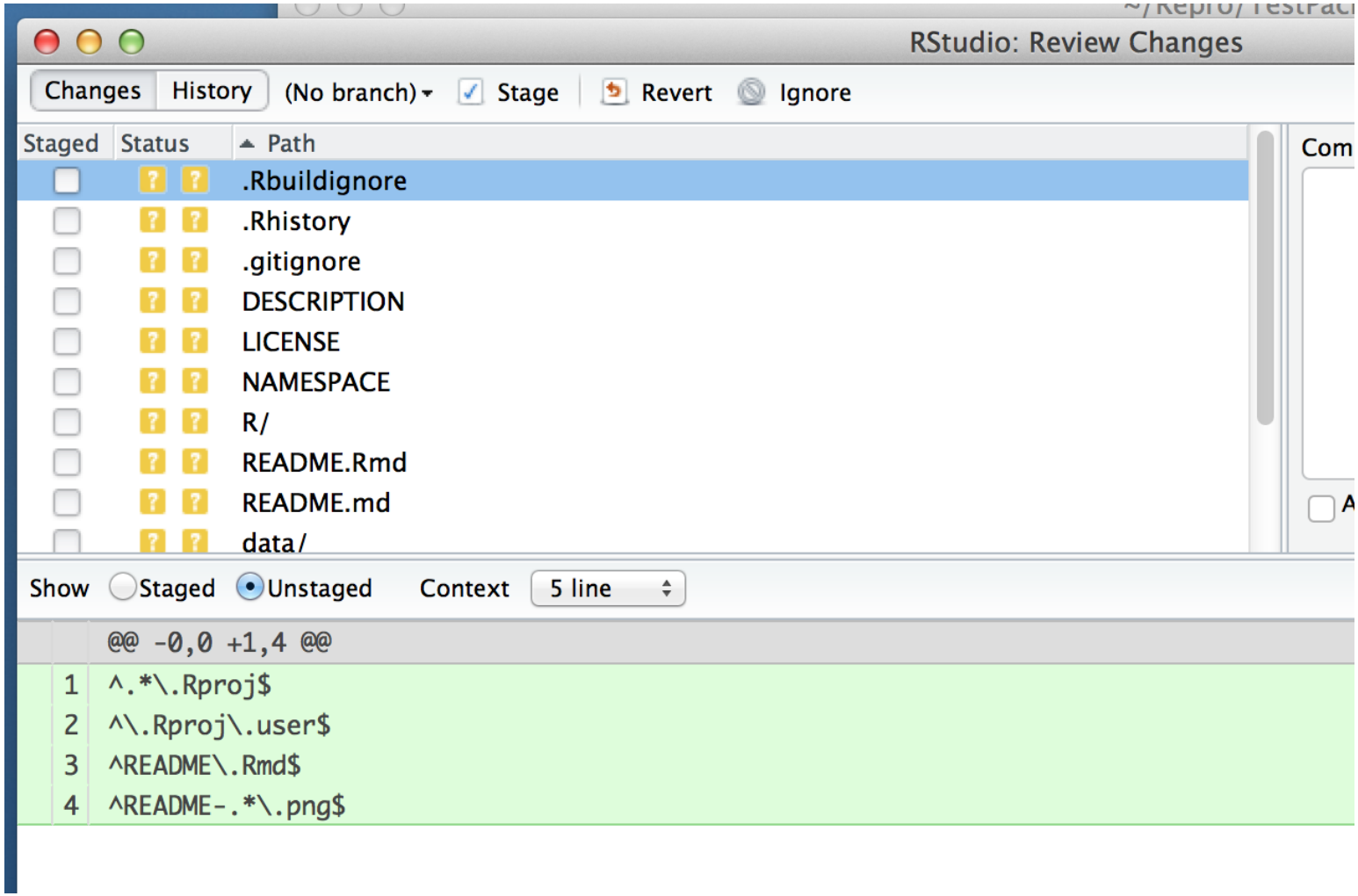
Git versioning proceeds in two steps.

First, you tell git a file has changes you'd like to track (you "add", or "stage" it)

Second, once you have enough tiny deltas amassed, you tell git to update the repository (you "commit" the set of changes).

At the start, adding changes means adding the whole file.

Stage and Commit - the Picture



You see this by clicking the “Commit” button

What Should I *Not* Include?

Guess what - *don't* version control Big Data in your repo.

Like R packages, git repos ideally have light footprints - you don't want them to be too big.

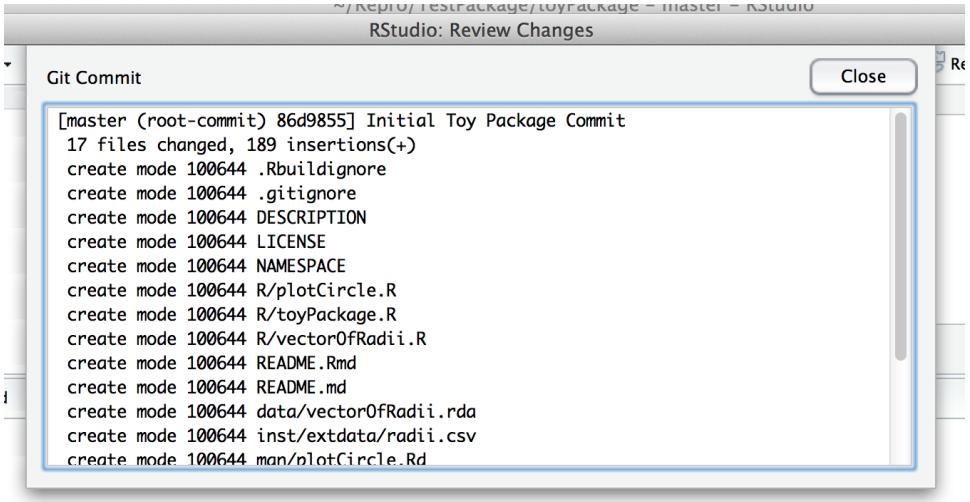
Also like R packages, that means we often don't want to include intermediate files (such as .aux from Latex), just the ones from which the final results can be derived (and in some cases the final products).

For our toy package, I'd include pretty much everything, with the exception of .Rhistory.

Guess What, Let's Commit

Once we've selected the files to add to our first commit, we need to add a brief message telling others (and our future selves) what we're doing.

The top message *line* should briefly state your purpose. If further explanation would help, leave a blank line after the first and add more. Here's the sign of success.



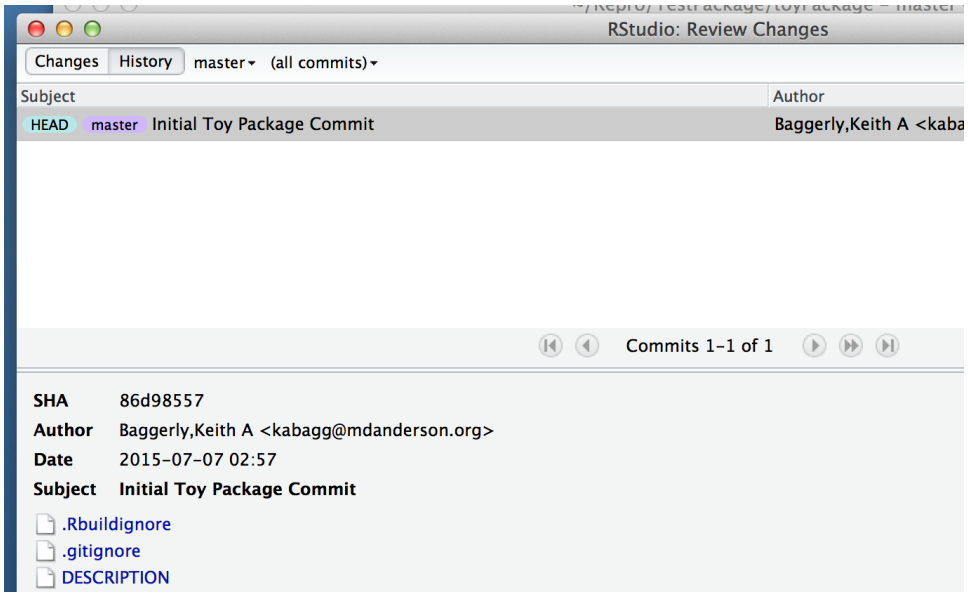
After the Commit

When we commit everything to the repository (and tell Rstudio to add the rest to .gitignore), the git pane goes blank.

It shows nothing because there's nothing that needs updating; all changes have been captured.

Checking the History

While there's nothing to add, there is now a history of edits, so you can see what changed. The history display will show new additions in green and new deletions in red, so you can look for "deltas" pretty quickly.



After the first commit, there's just a bunch of green.

The Story So Far

We've explored some of git's functionality and what it can let you do from the command line.

We've briefly explored init, add, commit, branch, and merge, and we snuck in some instances of status and show on the side.

We've pointed out how some of the most frequently used commands (init, add, commit, status) can be executed using Rstudio's graphic interface.

When we come back, we'll learn more about sharing with others, exploiting remote repositories, and GitHub.