Introduction to Git I

Lecture 3

Dirk Eddelbuettel

STAT 430: Data Science Programming Methods (Fall 2019)
Department of Statistics, University of Illinois

Recap

Shell

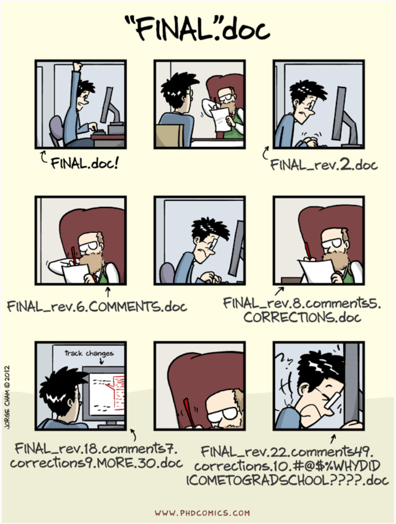
- We learned about a number of shell commands
- We learned that to Unix “everything is a file”
- We learned some simple scripting

Outline

Version Control (or ‘Revision Control’)

- This lecture and next lecture
- Very important (and somewhat tricky) topic
- This is “how stuff gets done”
- Knowing this is another core skill

Motivation



Source: http://phdcomics.com/comics/archive_print.php?comicid=1531

Funny only if have never had to work through a document sequence like this …

`git` helps with this problem.

Motivation

Why `git` ?

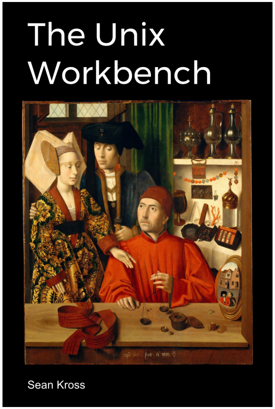
- We we will focus on simple key operations
- Git as a ‘time machine’
- Easy reversal to earlier versions
- Simple merge operations
- Pull request / cooperation / code review

Git tutorials

Free tutorials

- There are *lots* out there.
- One ranked list is at `https://hackr.io/tutorials/learn-git`
- One that looks promising `http://rogerdudler.github.io/git-guide/`
- Might be a good idea to try this, or another one
- We will also work very hands-on so follow along
- `git` is a *learning-by-doing* technology so if you can, *do*!

GIT READING



Please read Sections 6.1 to 6.4 from Chapter 6 on "Git and GitHub"

As before, not every line or paragraph is needed but there is decent overlap with what we cover here.

At `https://seankross.com/the-unix-workbench/`

Other Git reading

git - the simple guide - no deep shit! - Google Chrome

rogerdudler.github.io/git-guide/

git - the simple guide

just a simple guide for getting started with git. no deep shit ;)

Tweet

by Roger Dudler

credits to @tfnico, @fhd and Namics

in deutsch, español, français, indonesian, italiano, nederlands, polski, português, русски မြန်မာ, 日本語, 中文, 한국어 Vietnamese

please report issues on github

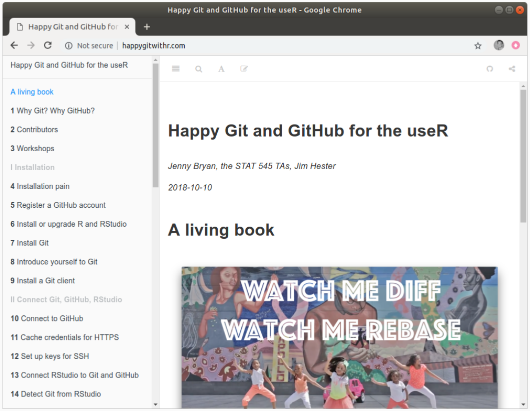
download the cheat sheet now. it's free!

want a simple but powerful git client for your mac?

Looks rather useful by *keeping it simple.*

At http://rogerdudler.github.io/git-guide/

Other Git reading


Happy Git and GitHub for the useR - Google Chrome

Happy Git and GitHub for

Not secure | happygitwithr.com

Happy Git and GitHub for the useR

A living book
1 Why Git? Why GitHub?
2 Contributors
3 Workshops
I Installation
4 Installation pain
5 Register a GitHub account
6 Install or upgrade R and RStudio
7 Install Git
8 Introduce yourself to Git
9 Install a Git client
II Connect Git, GitHub, RStudio
10 Connect to GitHub
11 Cache credentials for HTTPS
12 Set up keys for SSH
13 Connect RStudio to Git and GitHub
14 Detect Git from RStudio

Happy Git and GitHub for the useR

Jenny Bryan, the STAT 545 TAs, Jim Hester

2018-10-10

A living book

Focus on `git` from R / RStudio. Useful on installation etc (which we don't need because of RStudio Cloud).

Does not go far enough into `git`.

At `http://happygitwithr.com/` – also see `http://stat545.com/` (a well-known UBC course initially created by Jenny Bryan)

Git setup

Before the first use we need to set some configuration values:

```
## use *your name* and *your email* here
## use your netid (preferred) or gmail or ...
git config --global user.name "Your Name"
git config --global user.email "yournetid@illinois.edu"
```

This is needed because your (globally unique) email will identify your `git` actions once you share them. (Which we get to later.)

You should use the email address that registered with at GitHub as this email address will be used to associate your commits with you.

Git setup

If the shell terminal is unavailable you can do this from the R prompt as well relying on the `system()` function used to execute system (*i.e.* shell) commands:

```
system('git config --global user.name "Your Name"')
system('git config --global user.email "yournetid@illinois.edu"')
```

Note the outer `'` around the inner `"` – this is one way of quoting strings. Alternatively if you use double-quotes for the string you can escape the inner double quotes:

```
system("git config --global user.name \"Your Name\"")
```

Git Setup

Check it via

```
## check settings from previous page
## by inspecting beginning of my .gitconfig
head -5 ~/.gitconfig
```

```
## [user]
##  name = Dirk Eddelbuettel
##  email = edd@debian.org
## [push]
##          default = simple
```

(You will not see ## which is a side effect of Markdown rendering.)

Git setup

ILLINOIS

Alternatively, check it from R via

```
system("head -5 ~/.gitconfig")
```

Git setup: optional

Not required but helpful

- As we will see `git` operates with commands
- The typing can be tiresome so we can create 'shortcuts'
- These are called `aliases`.
- So you could copy the aliases from the next page is into `~/.gitconfig`, but it is not required.

Git setup: optional

Optional—but `git st` and `git ls` are something I do all the time.

Also at the website from where you can copy it.

```
[color]
    ui = true
[alias]
    st = status
    ci = commit
    co = checkout
    ls = log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset \
           %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit
    ll = log --pretty=format:\"%C(yellow)%h%Cred%d %Creset%s%Cblue [%cn]\" \
           --decorate --numstat
    hist = log --graph --decorate --pretty=oneline --abbrev-commit
    pu = pull --all --prune
## https://dev.to/devcamilla/why-git-alias-575h
## https://dev.to/megamattmiller/the-git-aliases-that-get-me-to-friday-1cmj
    nb = checkout -b
    fp = fetch -p
```

Note that the linebreak character \ is used just for display here. Use one line in your file.

A minimal git workflow

Start with

- `git add`
- `git commit`

to *add* a file to the staging area, and to then (after review, maybe) *commit* it to the repository,

then maybe

- `git revert`
- `git reset`

which can help to *revert* one (or more) commit, or to *reset* to a known prior state.

and always

- `git status`
- `git log`
- `git diff`

to check repo *status*, see what commits have been *logged* and *differences* between entities.

We will discuss all these commands in more detail in a moment.

Exercise

Try it!

- It is really important to practice `git`.
- You can do that on your computer
- Or in the RStudio Cloud instance we have access to
- There are also online tutorials that are useful see *e.g.* `https://try.github.io/`

First step

`git init`

- Sets up a `git` repository in the current working directory
- You can do this in an empty directory
- Or in one that already contains files
- Try it!

First step: `git init`

ILLINOIS

```
cd /tmp           # we operate in /tmp here
mkdir scratch     # create directory
cd scratch        # change into it
git init
```

```
## Initialized empty Git repository in /tmp/scratch/.git/
```

Second (and recurrent step)

`git status`

- This is a key commend
- We will do this a lot
- Asking `git` “what state are you in”
- Verifies success of previous command
- Sets stage for next command

Second (and recurrent step): `git status`

ILLINOIS

```
git status
```

```
## On branch master
##
## No commits yet
##
## nothing to commit (create/copy files and use "git add" to track)
```

(And with the aliases I showed earlier, `git st` is equivalent.)

Third step

git add

- Adds a file to the ‘staging area’
- The ‘staging area’ seems superfluous at first, it will grow on you
- We can stage multiple times to add, refine, …
- Here, we create a simple file, content does hardly matter
- We just use redirection here, feel free to use an editor instead
- `git status` tells us that we have a new file

Third step

```
echo "Some not so important content." > one_file.txt
echo "And some more." >> one_file.txt
git status
```

```
## On branch master
##
## No commits yet
##
## Untracked files:
##   (use "git add <file>..." to include in what will be committed)
##
##  one_file.txt
##
## nothing added to commit but untracked files present (use "git add" to track)
```

Third step

git add

- So we ‘add’ it
- `git add file` just adds one file, may be used repeatedly
- `git add -A` adds all files (not recommended as you may accidentally add files that should not be added; GUIs like the RStudio interface will help us select the ones we want)
- And ask `git status` again

Third step

```
git add one_file.txt
git status
```

```
## On branch master
##
## No commits yet
##
## Changes to be committed:
##   (use ”git rm --cached <file>...” to unstage)
##
##  new file:   one_file.txt
```

Fourth step

git commit

- This is actual transaction that adds to the repo
- Commiting requires a commit ‘log’ messages

Fourth step

```
git commit -m ”our first commit”
```

```
## [master (root-commit) c907e55] our first commit
##  1 file changed, 2 insertions(+)
##  create mode 100644 one_file.txt
```

The space after `-m` is optional: `-m”our first commit”` also works.

The ‘id’ of the commit will vary between sessions so you will not see the letters after `master (root-commit)` as seen here. That is normal.

Fourth step

```
git status
```

```
## On branch master
## nothing to commit, working tree clean
```

Fourth step

```
git log
```

```
## commit c907e554c23773c2d2a1092faa7a0b47807feeb4
## Author: Dirk Eddelbuettel <edd@debian.org>
## Date:   Sat Aug 31 16:22:38 2019 -0500
##
##     our first commit
```

Again, time and user and commit id will differ when you re-run this locally.

Review

- `git init`
- `git add`
- `git commit`
- `git log`

Another file

```
echo "One more" > second_file
git add second_file
git commit -m "Adding second file "
```

```
## [master b10c10d] Adding second file
##  1 file changed, 1 insertion(+)
##  create mode 100644 second_file
```

Another file

```
git log
```

```
## commit b10c10d1b3c8301743f06a17fdc8c4a9ff7236bd
## Author: Dirk Eddelbuettel <edd@debian.org>
## Date:   Sat Aug 31 16:22:38 2019 -0500
##
##     Adding second file
##
## commit c907e554c23773c2d2a1092faa7a0b47807feeb4
## Author: Dirk Eddelbuettel <edd@debian.org>
## Date:   Sat Aug 31 16:22:38 2019 -0500
##
##     our first commit
```

Modifying a file

```
echo ”A second line” >> second_file
git status
```

```
## On branch master
## Changes not staged for commit:
##   (use ”git add <file>...” to update what will be committed)
##   (use ”git checkout -- <file>...” to discard changes in working directory)
##
##  modified:   second_file
##
## no changes added to commit (use ”git add” and/or ”git commit -a”)
```

Modifying a file

```
git diff
```

```
## diff --git a/second_file b/second_file
## index dab5687..a6e1a10 100644
## --- a/second_file
## +++ b/second_file
## @@ -1 +1,2 @@
##  One more
## +A second line
```

Modifying a file

```
git add second_file
git commit -m "expand file two"
```

```
## [master eb550b5] expand file two
##  1 file changed, 1 insertion(+)
```

Modifying a file

ILLINOIS

```
git log
```

```
## commit eb550b52fef5a920c86688a9a56fd76ed584fdd9
## Author: Dirk Eddelbuettel <edd@debian.org>
## Date:   Sat Aug 31 16:22:38 2019 -0500
##
##     expand file two
##
## commit b10c10d1b3c8301743f06a17fdc8c4a9ff7236bd
## Author: Dirk Eddelbuettel <edd@debian.org>
## Date:   Sat Aug 31 16:22:38 2019 -0500
##
##     Adding second file
##
## commit c907e554c23773c2d2a1092faa7a0b47807feeb4
## Author: Dirk Eddelbuettel <edd@debian.org>
## Date:   Sat Aug 31 16:22:38 2019 -0500
##
##     our first commit
```

Modifying a file

This log is similar to the one from my aliases, but without color in order to be typset here:

```
git log  --graph --pretty=format:'%h %d %s (%cr) <%an>' --abbrev-commit
```

```
## * eb550b5  (HEAD -> master) expand file two (0 seconds ago) <Dirk Eddelbuettel>
## * b10c10d  Adding second file (0 seconds ago) <Dirk Eddelbuettel>
## * c907e55  our first commit (0 seconds ago) <Dirk Eddelbuettel>
```

GIT LOG, AND COMMIT IDS

ILLINOIS

`log`

- The appreviated log showed the commit identifiers
- These are are SHA-1 checksums (which vary in repeated runs)
- We can refer to commits
    - by the sha-1
    - or by their *relative position*
- Most recent is always `HEAD`, preceding `HEAD~1` and so on

GIT REVERT

`revert`

- Commits can be reverted: `git revert`
- *Adds a new commit* (*i.e.* transaction) which undoes the change
- *I.e.* it is not “rewinding” but explicitly reverting
- Try
  - `git revert HEAD -n; git commit -m 'revert once'`
  - then view the content of the file changed: `cat second_file`
  - revert again (!!) with another `git revert HEAD -n; git commit -m 're-revert'`
  - and view the file again
  - then view the log

GIT RESET

ILLINOIS

`reset`

- Another method for ‘getting back’ to a previous stage
- `git reset COMMITID` does a soft reset:
    - it “unwinds” the commit, and
    - keeps the changes, so `git status` shows modified file
    - a good first step
- `git reset --hard COMMITID` does a hard reset:
    - restores state at the named commit
    - status as at the commit so all changes unwound

where COMMITID is the SHA-1 identying a commit eg in the log.

GIT DIFF

`diff`

- Comparison across time (and more once we get to branches)
- `git diff HEAD` compares to committed files
- `git diff file` compares differences just in the named file
- `git diff --cached commit` compares to the (staged) index

Useful Git Interface

- you need to be in a 'Project'
- try creating a project
- try making it a `git` repo
- repeat the steps above about
    - write a file or two, then change it
    - adding it to `git`, and then commit it to `git`
    - seeing the log
    - changing the file
    - etc pp
- experiment!

Next lecture: More advanced git

ILLINOIS

Outlook

- branching
- changing branches
- pull request with oneself
- pull request with others
- GitHub