How to use `git` for version control

Why learn git?

"Move fast and break things" mentality.

Multiple versions of files without having multiple files.

Reproducibility.

Open science.

Verify that git is installed and configured

Open a terminal and type the following commands to verify that your installation of git is installed and configured properly.

```
git config --help  # read some of the help text
git config --list  # list the configured variables
```

You should see your `user.name` and `user.email` in the output. If not, follow the steps in this video[1].

[1] git-scm.com/video/get-going

Configuring `git`

- The only required steps are to define a `user.name` and `user.email`.
- The “git user” is the author of the changes you make.

Why don’t you need to define a `user.password` configuration variable to use `git`?

Configure `git` to use a text editor

When using `git`, you will need a text editor to write commit messages.

You should learn how to use a command line text editor like nano or vim, or download a text editor that `git` can be configured to work with.

```
# configure git to use the text editor "atom"
git config --global core.editor "atom --wait"
```

See this help article[2] for instructions on how to configure `git` to use other common text editors like Sublime, TextMate, and Notepad++.

[2] help.github.com/en/articles/associating-text-editors-with-git

Create a new git repo

Move to the Desktop or any place you'd like to create a new git repo. You will delete it at the end of this section.

```
cd ~/Desktop/
git init proj1  # start a new repo named "proj1"
ls proj1        # why was nothing created?
rmdir proj1     # why is the directory not empty?
ls -la proj1    # what was created?
```

What's inside the ".git" directory?

If you are lucky, you will never have to poke around in here.

```
proj1/.git
├── HEAD
├── config
├── description
├── hooks
│   └── (11 files, sample hooks)
├── info
│   └── exclude
├── objects
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags
```

Running `git status`

`git status` will be one of your most used commands.

The presence of a valid ".git" directory is how git knows to watch the contents of this directory for changes.

Run the `git status` command from outside and inside the "proj1" directory and observe the differences.

```
cd ..         # move outside the proj1 directory
git status    # try to git status outside a repo
cd proj1      # move in the repo
git status    # check the status of an empty repo
```

Your first commit

Create an empty text file in the proj1 directory.

```
touch first-file.txt  # create an empty file
git status            # run this command after each step
```

Follow the steps described in the output of the git status command to include the new file “first-file.txt” in what will be committed:

```
git add first-file.txt
git status
```

Writing a commit message

Create a commit by entering a commit message:

```
git commit  # opens an editor to write the commit message
```

Note that `git` will wait in the terminal until you write a commit message, save, and close the file.

Writing a commit message from the command line

- Alternately, you can enter the commit message from the command line.
- Best for small changes that can be described in a few words.
- To describe larger changes you will want to use a text editor.

```
git commit -m "Added an empty text file"
```

All commits must have messages. This is a fundamental difference between “committing” and “saving”.

Viewing the commit history

After creating your first commit, run the following commands, and try to understand the output.

```
git status
git log
git log --oneline
```

Starting a second commit

Create two more empty files to commit to the repo.

```
touch second-file.txt third-file.txt
```

Continue to run `git status` after each step to verify what is going on.

Methods for adding files

Add the files using one of the following methods:

```
# Method 1
git add second-file.txt
git add third-file.txt

# Method 2
git add second-file.txt third-file.txt

# Method 3
git add *-file.txt

# Method 4
git add .
```

Unstaging changes you do not want to commit

Use `git reset` to move changes back and forth between the Working Directory and the Staging Area.

```
git reset second-file.txt
git status
git add second-file.txt
git status
```

You will most often need this when you run `git add .` and realize you added something you did not want to.

Authoring the second commit

Commit the two new files to the repo:

```
git status  # make sure the two files are staged
git commit -m "Add two new empty files"
git status
git log
```

Deleting a file

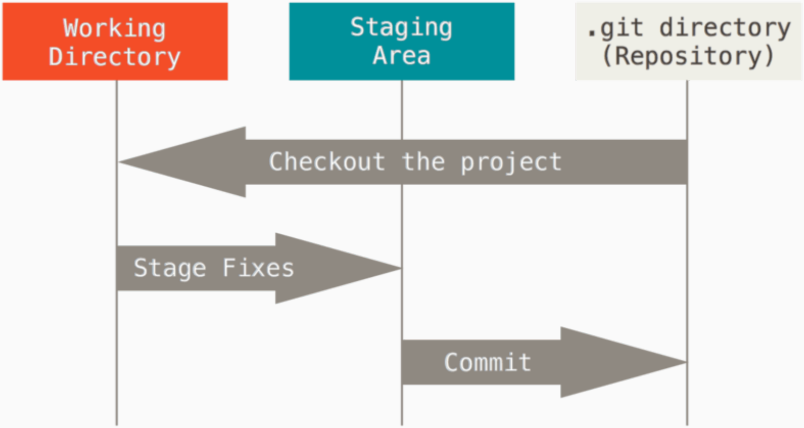
To delete a file from a git repo, you need to mark it as purposefully removed with `git rm [file]`.

```
rm second-file.txt     # remove the file
git status
git rm second-file.txt # mark as purposefully removed
git status
git commit -m "Removed the second file"
git log
git log --oneline
```

You don't have to run both `rm [file]` and `git rm [file]`.
`git rm [file]` will remove the file too, if it exists.

git areas

Once a change is committed to the git database, it is considered permanent.
It is possible to undue things that are permanent, but it's harder.



Go back in time

With git we can always roll back time to whatever commit we want.

```
git log --oneline
# copy the SHA of the commit with the message:
# "Add two new empty files"
git checkout [SHA]  # roll back to that commit
ls  # notice the deleted file "second-file.txt"
git log
git checkout master  # go back to present
ls  # notice "second-file.txt" has been deleted
git log
```

Get a file you once deleted

Using the same SHA from the previous step, provide an additional argument to `git checkout` to retrieve a file you once deleted.

```
git checkout [SHA] second-file.txt
git status
git commit -m "Revived second file"
```

Creating a presentation

Remove all empty files and create a new one for a presentation.

```
git rm *-file.txt
git commit -m "Removed all empty files"
touch presentation.Rmd
```

Rmarkdown presentation

Create a simple Rmarkdown presentation file with the following contents. **Note:** Replace the quotes with backticks.

```
---
title: Detecting changes in change detection detection
output: beamer_presentation
---
'''{r config, include=FALSE}
knitr::opts_chunk$set(echo=FALSE, cache=TRUE)
library(tidyverse)
'''
```

Add the presentation to the repo.

```
git add presentation.Rmd
git commit -m "Add the title slide for the presentation"
```

Compile the Rmarkdown presentation

Compile the presentation by opening it in RStudio and pressing "knit", or running the following R command:

```
Rscript -e "rmarkdown::render('presentation.Rmd')"
```

You should see a new file, "presentation.pdf". Open it to see your presentation.

```
git status
open presentation.pdf
```

Files to ignore

We want git to keep track of changes to "presentation.Rmd" but we don't want it to keep track of changes to pdfs, like "presentation.pdf".

To ignore files, add them to a new file called ".gitignore" with the following contents:

```
# contents of .gitignore
*.pdf
```

Then commit the new file .gitignore to the repo.

```
git status  # what happened to presentation.pdf?
git add .gitignore
git commit -m "Ignore pdfs"
```

Add an exception

To add an exception to the rule, for example, to save a published version of the pdf, you can force git to add the file, or add an exception to the .gitignore.

```
mkdir talks
cp presentation.pdf talks/CogSci20.pdf

# Method 1: Force git to track the file
git add -f talks/CogSci18.pdf

# Method 2: Add an exception to the .gitignore
# for any pdfs in "talks/"
echo "!talks/*.pdf" >> .gitignore

# Commit the changes
git commit -m "Add slides for talk given at CogSci20"
```

Tracking changes to files

Add a new slide to the presentation. Remember to use backticks instead of quotes.

```
# Results


'''{r reaction-times}
m <- 0.06
b <- 2.1
df <- tibble(
  x = 1:100,
  y = m * x + b + runif(100)
)
ggplot(df) +
  aes(x, y) +
  geom_point()
'''
```

Reviewing differences

Run `git diff` to view the changes you made to "presentation.Rmd"

```
git status
git diff
git add presentation.Rmd
git status
git diff
git diff --cached
```

Commit the new slide

```
git add presentation.Rmd
git commit -m "Add results slide"
git log --oneline
```

Compile the presentation again

Now that our presentation has chunks that are cached and generated figures, we have additional files to ignore.

```
Rscript -e "rmarkdown::render('presentation.Rmd')"
git status
echo "*_cache/" >> .gitignore
echo "*_files/" >> .gitignore
git status
git add .gitignore
git commit -m "Ignore cache and output files for Rmd"
```

Creating a new branch

Branches are great! Use them.

- Branches are bifurcations in the change history of a project.
- The default branch name is named “master” branch by convention.
- To switch branches, you will use the `git checkout` command.

Create a new branch and switch to it.

```
git status  # Note the first line of the output
git branch
git branch --help
git branch new-intro
git status  # Still on master...
git checkout new-intro
git status
```

Shortcuts

To create a new branch and switch to it in one command, run the following:

```
git checkout --help  # read about the "-b" flag
git checkout -b new-intro
```

Working on the new intro

Edit presentation.Rmd to add a new intro slide.

```
# Intro


How many changes does it take
to change a change detector's
change detection rate?
```

Add the change. Note the shortcut `git commit -am ...` which is short for `git add . && git commit -m ...`

```
git status  # should see change to presentation
git commit -am "Add a motivating question"
```

Merging

- Eventually you will want your branches to converge.
- Technically, one branch will merge with another one.
- Order matters! Merging branches A into B may produce different results than merging B into A.
- In many cases, git handles merge conflicts intelligently.
- In the worst case, you have to handle a merge yourself.

Simulating a merge

```
git status  # On branch new-intro
```

Edit the results plot on the new-intro branch.

```
ggplot(df) +
  aes(x, y, size = error) +
  geom_point(shape = 1) +
  geom_smooth(method = "lm", se = FALSE,
              show.legend = FALSE)
```

Commit the changes to the plot.

```
git diff
git commit -am "Scale points by error and add regression li
```

Add changes on the master branch

```
git checkout master
```

Edit presentation.Rmd to adjust the plot, simulating the merge conflict.

```
ggplot(df) +
  aes(x, y) +
  geom_point(alpha = 0.4) +
  geom_smooth(method = "lm", se = FALSE,
              show.legend = FALSE)
```

Add a conclusion slide

While still on the master branch, also add a Conclusion slide that is independent of the plot.

```
# Conclusion

It takes 100 changes to
change a change detector's
change detection rate!
```

Commit the changes on the master branch.

```
git status
git commit -am "Update the plot and add conclusion"
```

Your first merge conflict

We are going to merge the master branch into the new-intro branch.

```
git checkout new-intro
git merge master
# Auto-merging presentation.Rmd
# CONFLICT (content): Merge conflict in presentation.Rmd
# Automatic merge failed; fix conflicts and then commit the
```

Open the file in your text editor or RStudio and look at the file.

Merge conflicts

You should see that git has made presentation.Rmd look like this:

```
ggplot(df) +
<<<<<<< HEAD
  aes(x, y, size = error) +
  geom_point(shape = 1) +
=======
  aes(x, y) +
  geom_point(alpha = 0.4) +
>>>>>>> master
  geom_smooth(method = "lm", se = FALSE,
              show.legend = FALSE)
```

Understanding the merge conflict

- `<<<<<<< HEAD` marks the start of a merge conflit
- HEAD is whatever branch you are on (in this case, “new-intro”)
- `>>>>>>> [branch]` marks the end of a merge conflict, where `[branch]` is the name of the branch you are merging in.
- In this case, we are merging the master branch into the new-intro branch.

Things `git` did automatically

- `git` did not care about the new intro or conclusion slides. It merged these parts automatically.
- `git` also didn't care about the `geom_smooth` call which was independently added on both branches.

Aborting the merge

Run `git status` and figure out how to abort the merge.

After aborting the merge, merge from the other side.

```
git checkout master
git merge new-intro
```

Open presentation.Rmd and read the merge conflict. It should make sense to you now.

Fix the conflict

Replace the conflict (including `<<<<<<< HEAD` and `>>>>>>> new-intro`) with a merged version:

```
ggplot(df) +
  aes(x, y, size = error) +
  geom_point(shape = 1, alpha = 0.4) +
  geom_smooth(method = "lm", se = FALSE,
              show.legend = FALSE)
```

Mark the merge as completed.

```
git add presentation.Rmd
git commit  # note the different default message
git log
```

Collaborating

To allow others to collaborate on a project, we can make the repo available on a site that allows free hosting of repos, like GitHub, GitLab, or BitBucket.

To collaborate on someone else's project, use the `git clone ...` command:

```
cd ~/Desktop
# clone the repo containing these slides
git clone https://github.com/pedmiston/git-and-github.git
cd git-and-github
open slides/version-control.pdf
```

Why don't you need to provide a GitHub username or password to download these materials?

Create a private repo and clone it

- Login to github.com, select “New repository” from the “+” menu.
- Give your repo a dummy name like “private-proj”
- Select the radio option to make the repo “Private”
- Check the box “Initialize this repository with a README”.
- Select the button to “Create the repository”

Then clone the newly created repo to your computer.

```
cd ~/Desktop
git clone https://github.com/[username]/[private-proj].git
# now you should have to provide your GitHub credentials
cd [private-proj]
cat README.md
git status
```

Creating an endpoint for your local repo

Now we will configure your existing proj1 to push to GitHub.

- Login to github.com, select “New repository” from the “+” menu.
- Give your repo a temporary name, like `proj1`.
- Select the radio option to make the repo Public.
- For the option “Initialize this repository with a README”, note the question says “Skip this step if you’re importing an existing repository.” Since we are importing an existing repository, you should not check this box.

Pushing your repo to GitHub

After creating the repo, follow the steps for “push an existing repository from the command line” on the newly created repo page.

```
cd ~/Desktop/proj1
git remote add origin https://github.com/[username]/proj1.g
git push -u origin master
```

After running the `git push ...` command from the terminal, refresh the page, and you should see your repo online for someone to clone.

Brace yourselves!

Pro Git

Oh, shit! Git