A crash course in version control with git

Hannah Holland-Moritz

February 18, 2020

Overview

Resources/Links/Inspiration:

Today's presentation is heavily inspired by:

1. Sofware Carpentry's lesson in git
   - https://swcarpentry.github.io/git-novice/index.html
2. Max Joseph's git intro presentation
   - https://github.com/mbjoseph/git-intro
3. Visual Git Reference
   - http://marklodato.github.io/visual-git-guide/index-en.html
4. git - the simple guide
   - https://rogerdudler.github.io/git-guide/
5. Think like (a) Git (good site for advanced beginners - that's you, after today!)
   - http://think-like-a-git.net/

Today's Topics

1. Introduction to `git`
2. Git in the command line
   - First steps
   - Setting up repositories
   - Working in repositories
     - The change -> add -> commit cycle
   - Tracking Changes
   - Ignoring files with `.gitignore`
   - Using Github (and other remotes)
   - Collaborating
   - Conflicts
3. Git in RStudio

Introduction to `git`

Why should we be using version control?

1. To keep track of changes
   - Avoid the `mypaper_final_final_reallydonethistime.docx` problem.
2. To document reasons for each change.
3. To preserve multiple versions of documents simultaneously.
   - This can be done with "branches".
4. To collaborate with others and not create conflicting versions of the same document.

What is `git` doing?

- Git keeps track of your changes. It monitors changes as if they were separate from the document itself.
- Each change is a snapshot of the project in it's current state.
  - you get to choose which files are in the picture
  - you can compare your picture to collaborators' pictures and choose to accept changes you like.
- Git commands take the format `git verb options`

`git` in the command line

Setup

Make a new directory

```
mkdir microbes
cd microbes
nano test.R
```

Write some short code

```
x <- rnorm(n = 50, mean = 5, sd = 1)
saveRDS(x, "x.RDS")
```

To exit `nano` type:

`ctrl+o`, `enter` (this saves the document)

`ctrl+x`, `enter` (this closes the editor)

Now we're ready to begin...

The first time you use git

- You will need to tell git who you are so it knows who made the changes in your documents.
- To do this, we set a user name and user email.

```
git config --global user.name "Mickey Mouse"
git config --global user.email "mickey1234@gmail.com"
```

- Check your configuration with the following command:

```
git config --list
```

The first time you use git

Optional:

- Change core editor to `nano` from `vim`

```
git config --global core.editor "nano -w"
```

Click here for more config options

Setting up a repository

repository: *a storage area (usually a directory) where git can store all the history of a project and information of who changed what and when.*

1. Make (*initialize*) a new repository

```
cd microbes
git init
```

Setting up a repository

Check the repository was created

```
git status
ls -a
```

You'll see a message about the branch, files that are committed/uncommitted, the commits, and a list of the files including the `.git` file.

Working in a repository

The git workflow

1. make changes to file(s)
2. "stage" those file(s)
3. commit the changes

Worksheet time!

Working in a repository

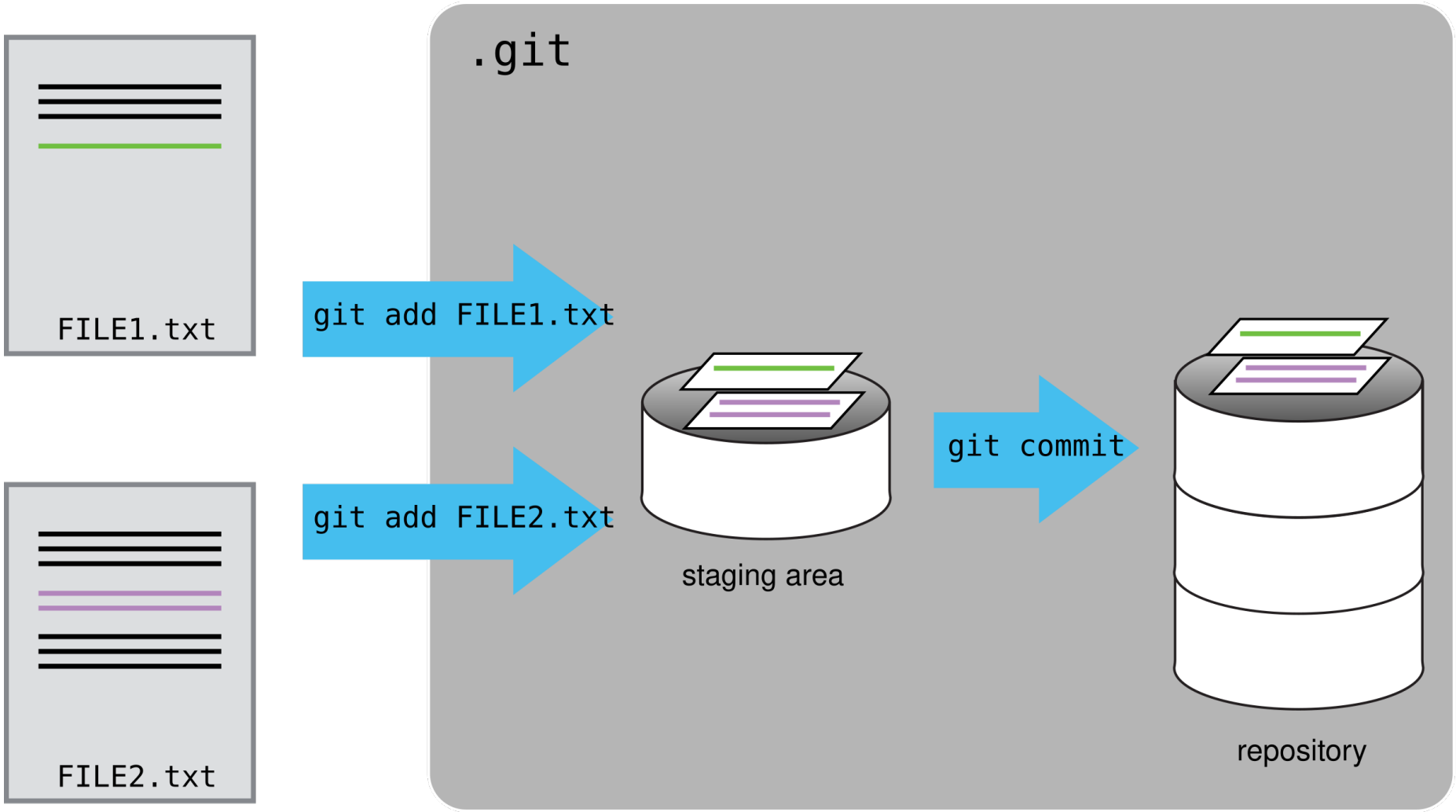

Figure 1: “the git workflow”

Working in a repository

Now let's do this ourselves:

Working in a repository

Now let's do this ourselves:

Check on git status

```
git status
```

One file - our `test.R` - file is "untracked".

Working in a repository

Now let's do this ourselves:

Check on git status

```
git status
```

One file - our `test.R` - file is "untracked".

Stage the file

```
 git add test.R
```

Our file is now *staged* and ready to be committed.

Working in a repository

Now let's do this ourselves:

Check on git status

```
git status
```

One file - our `test.R` - file is "untracked".

Stage the file

```
 git add test.R
```

Our file is now *staged* and ready to be committed.

Commit the changes

```
git commit test.R -m "created initial test.R file"
```

- We *commit* (take a snapshot of) the changes
- and give a message (`-m` flag) explaining what the purpose of the changes was.

Working in a repository

An aside about commit messages

- like a lab notebook - should be informative
- ideally, you should commit often and in small parts. This makes reverting back easier

Bad messages:

```
-m "some updates"
-m "fixes bugs"
-m "adds three new sections"
```

Working in a repository

Now it's your turn:

1. Create a new file called `plan_for_world_dominion.txt`
2. Write the following line for `plan_for_world_dominion.txt`

```
Microbes rule the world.
```

3. `add` and `commit` the file.

Tracking changes

- How can you figure out the differences between two files?
- How do you go back in time, to a previous version of a file(s)?

Tracking changes

First...

Add some code to our test.R file:

```
x <- rnorm(n = 50, mean = 5, sd = 1)
saveRDS(x, "x.RDS")
y <- rnorm(n = 50, mean = 1, sd = 1)
saveRDS(y, "y.RDS")
```

```
git status
```

The file is now shown as modified, but not yet staged.

Tracking changes

What if we forgot exactly what changed?

```
git diff # shows line-by-line changes
```

Fancy versions of `git diff`

```
git diff --color-words # shows word-by-word changes
git diff --staged # shows changes when a document is staged
```

Tracking changes: Looking back in time

Looking back in time

```
git log
```

- The log shows a history of the commits you've made. Each is given a unique identifier that you can use to refer to that point in the history.
- For simplicity, git allows you to use the first 7 characters to refer to the entire identifier.

Tracking changes: Going back in time

- Everytime you make a commit, git takes a snapshot of all the changes to your committed files
  - Each snapshot is referred to as a *HEAD*
- Your history (`git log`), is a stack of HEADs
- You can navigate between HEADs (i.e. versions) using the `git checkout` command

Worksheet time!

Tracking changes: Going back in time

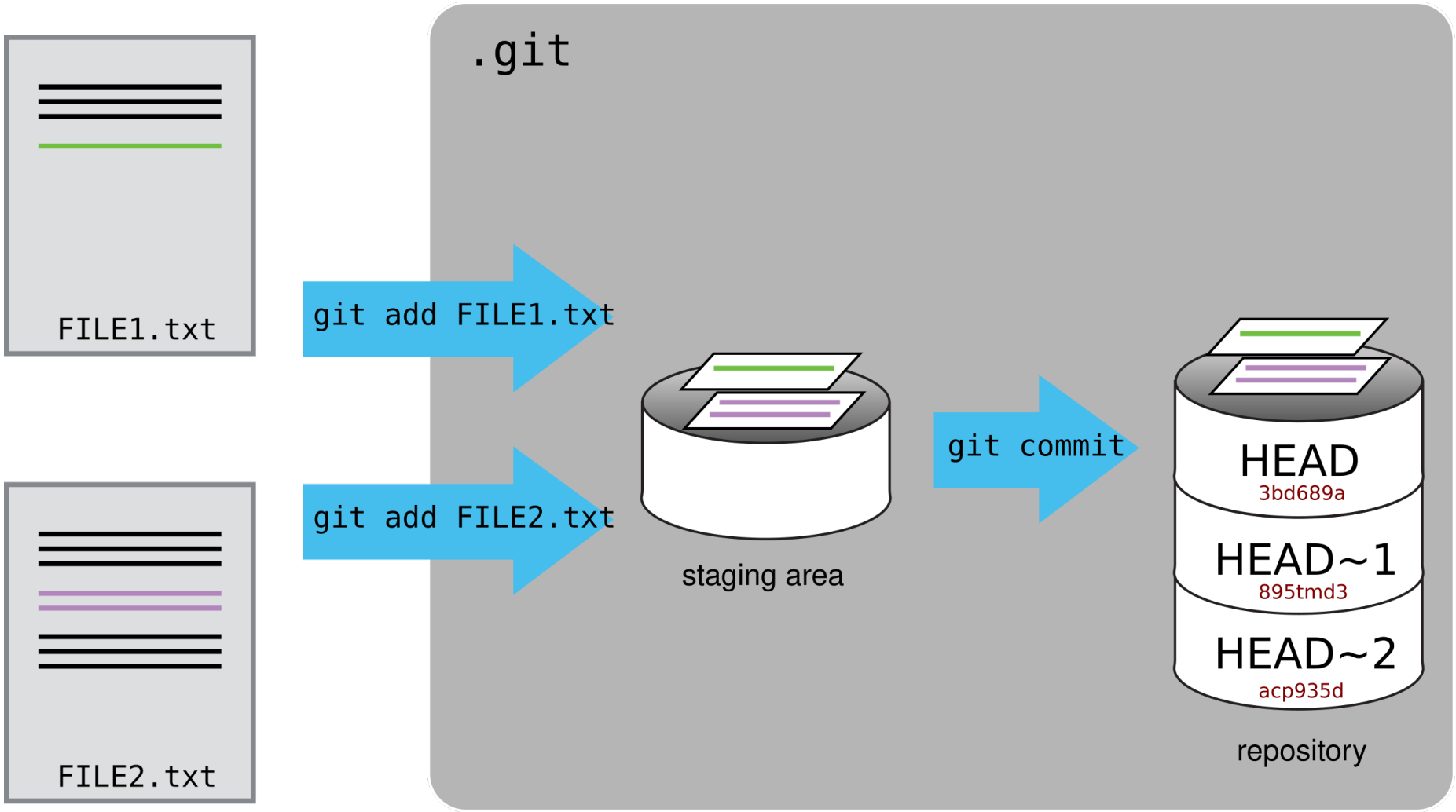

Figure 2: "the git workflow"

Tracking changes: Going back in time

Going back in time

```
git checkout HEAD~1 test.R
```

- The `~1` refers to the number of steps backward you want to go.
- If you don't want to count backwards, you can also use the unique identifier from `git log`

```
git checkout fe452Eu test.R
```

Going back in time

If you go back in time using the `git checkout HEAD~1 <file>` command, your file changes will register as modified, as if you just implemented the changes but have not yet staged or committed them. You can make changes, and proceed with the changes -> add -> commit cycle.

Tracking changes: Going back in time

Your turn!

1. Use `nano` to add the following line to the end of test.R

```
This line is a terrible idea.
```

2. Add and commit the change (don't forget to add a commit message!)
3. Use `git checkout` to undo the change.
4. Replace the line with a better line

```
# This line is a much better idea
```

5. Add and commit the change.

Tracking changes: Going back in time

WARNING! - the detached HEAD state

`git checkout` has multiple functions. If you don't specify a file name after `git checkout`, you will go backwards in time, but not be able to commit any changes that you make. This is known as a *detached HEAD* state.

- To get out of a detached HEAD state without saving any changes

```
git checkout master
```

Ignoring files

Git allows you to ignore files that you don't want tracked. Often these files are very large, temporary, or unnecessary to generate the final product.

Examples of good files to ignore:

- RStudio files: `.Rproj.user`, `.Rhistory`, `*.Rproj`, `.RData`
- `knitr` cache files `*_cache/`
- Sensitive data files

To ignore these files use `nano` to create a file called `.gitignore` and add them in a list.

Ignoring files

Example:

```
[hannah@localhost microbe]$ cat .gitignore
.Rproj.user
.Rhistory
microbes.Rproj
.RData
```

Ignoring files

Your turn!

1. Use `nano` to create a file called `to_do.txt`. You don't need to keep track of changes in your to-do list so you will add it to your `.gitignore` file.

```
nano to_do.txt
```

```
[hannah@localhost microbe]$ cat to_do.txt
* write a plan
* execute plan
* celebrate
```

2. Use `nano` to create a file called `.gitignore`
3. Add `to_do.txt` to the `.gitignore` file and then save and exit.
4. Check your `git status`.

Break

Working on branches

- Branches allow you to create and test updates without affecting the working copy.
- The default branch is called `master`
- You probably want to create a new branch if:
  - You are working on a series of large changes that constitute one update.
  - You want multiple working versions of your code (for example, a first and second draft for a paper)
  - You think you might want to easily refer back to that repository state at some time in the future.

The basic branch workflow: 1. Create a new branch 2. Make changes, test, commit those changes. 3. "Merge"" the branch with the original copy of your work (the `master` branch)

Working on branches

To create a branch:

```
git branch # check what branch you are currently on
git checkout -b <branchname> # the -b creates the branch; chec
git branch # see that you have switched to the new branch.
```

To switch between branches:

```
git branch # check what branch you are currently on
git checkout <branchname> # checkout moves you to the branch
git branch # see that you have switched to the new branch.
```

Working on branches

Your turn!

1. Create a new branch called `development`, and switch into it.
2. While on `development` make a new file called `notes.txt`; Write a note to yourself inside the file using `nano`.
3. Add and commit the changes. Check your `git status`
4. Type `ls`
5. Now changes branches back to master, and type `ls` again. What do you notice?

Working on branches: Merging

Many people dread merging because it often leads to conflicts. Conflicts happen when `git` can't figure out on it's own how to merge two files.

- Merge conflicts usually happen because:
- Two or more people make different changes to the same line of the same file.
- One person deletes a file, and another person makes changes to the same file.

Luckily branches provide a great solution for this! You can test out a merge before it happens and if all goes well, you can run the merge for real.

The Scout Pattern - this method attributable to think-like-a-git.net

1. Move to the master branch (or whatever branch you want to merge with)

```
git checkout master
```

2. Create a new branch to test the merge and switch to it.

```
git checkout -b test_merge
```

Using Github (and other "remotes")

remote: *a repository that is not on your local computer. Remotes can be on any number of places, including websites like GitHub or Bitbucket, and on remote servers, such as microbe/proteus.*

- Useful if you want a master copy that everyone draws from and contributes to.

Create and connect to a remote

```
git remote add origin https://github.com/hhollandmoritz/microb
git push -u origin master
```

What are origin and master????

- "origin" refers to the remote repository; technically you can call it anything, but most git documentation uses this convention so for clarity we will too.
  - "origin" is confusing because often you are making changes locally and then moving them to the remote repository rather than the other way around - unless you are collaborating.
- master refers to the branch

Connect to an already-existing remote:

Cardinal Rule: Always `pull` before you `push`

```
git push origin master
         ^      ^
         ^      name of the branch on the remote
         name of the remote
```

Collaborating

https://github.com/hhollandmoritz/collaboration_practice

Your turn!

1. Use nano to create a text file with your name, and write your favorite food in the text file.

```
nano hannah.txt
cat hannah.txt
>chocolate
```

2. Add and commit your file.
3. Pull any changes from the remote repository first.
4. Push your changes to the repository.

Conflicts

Git in RStudio

Extras

Changing editors

- For more editor options see https://swcarpentry.github.io/git-novice/02-setup/index.html)