Version control with Git tutorial

Experiential Data science for Undergraduate Cross-disciplinary Education

Dr. Kim Dill-McFarland, U. of British Columbia

Contents

Version control with Git tutorial

Learning objectives

- Practice Unix command line
- Enact version control on a text file using Git command line tools

Setup

If you would like to follow along:

1. Download the Git installer for your operating system
   - If you are using the GitBash terminal, you already have git installed
2. Run the installer and follow prompts
3. Check that git is installed by opening your terminal and inputting `git`[Enter]
   - You should see a help page starting with

     ```
     usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
        [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
        [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
        [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
        <command> [<args>]
     ```
   - You may need to restart your computer to fully install
   - If you do not have a Unix terminal, see the 'Download_Unix_terminal' instructions.

Version control

Version control is any means of recording changes to file(s) over time so that you can recall or revert back to specific versions later.

One method of version control you may have employed in the past is dated or versioned file names like `myFile_2018.06.25.txt` or `myFile_v1.2.txt`. While these systems can get the job done, you end up will a

long list of files that take up storage space and have little information on how they differ. It remains up to you to remember what date or version to go back to for whatever purpose is needed.

Why Git?

There is a better way to version control with distributed systems, and Git is one such language/program. While Git can be used to track changes for any file type, it is most useful with text-based files like `.txt`, `.Rmd`, `.R`, etc. because these are *source files* with the appearance directions clearly noted and not hidden behind a GUI. Think of Git like the "track changes" feature of MS Word, except there is a record of the changes after you "accept" them.

The advantages of distributed version control over naming methods are that:

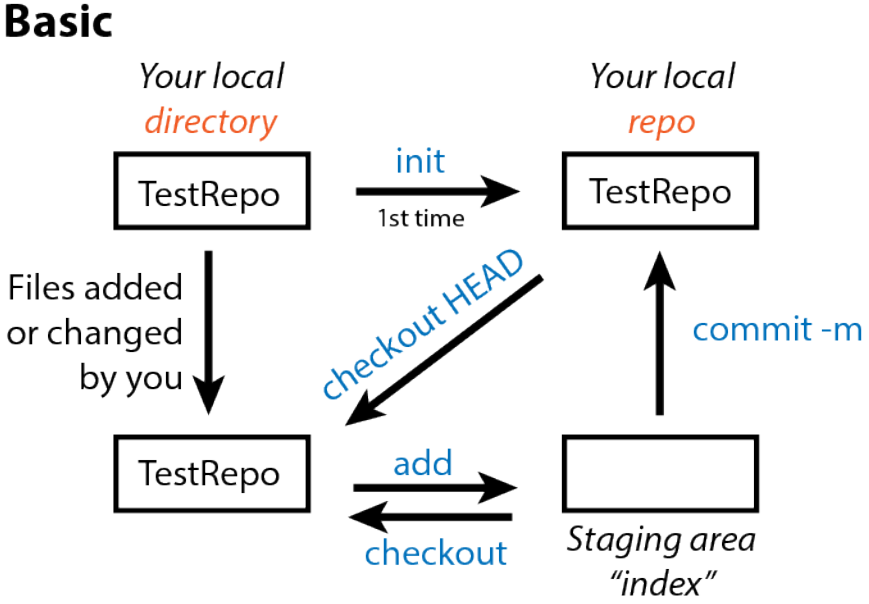
- it is easy to see what changed when (time-stamped), why (commented), and by whom (author identified)
- you can jump back to any point in time since the file's creation, not just versions you deemed important enough to save at the time
- you have the freedom to experiment, try something crazy even, because you can always go back to the last known good version
- you can work on 2+ different versions in parallel
- you are able manage contributions from multiple people (see later section on GitHub)

Specifically, we are using Git, as opposed to another program, because it:

- is very fast, even for projects with hundreds of files and years of history
- doesn't require access to a server
- is already extremely popular

Using Git

All Git functions are accessed from the command line with `git` followed by the function name like `git init`. You do not need to open a specific program other than your terminal. The commands used in this section are summarized in the following infographic. Please refer to it as we progress through the examples.



Initialize a repo (`git init`)

Git will not implement version control on your files until you tell it to do so. This is accomplished by initializing a repository with `init`.

First, create a directory on your desktop named `TestRepo`. Then, open your terminal and navigate to the `TestRepo` directory.

```
mkdir Desktop/TestRepo
cd Desktop/TestRepo
```

Initialize the TestRepo directory with

```
git init
```

```
## Initialized empty Git repository in /Users/kim/Desktop/TestRepo/.git/
```

If you now list all the files in the `TestRepo` directory with `ls -a`, you will see a `.git` file that designates that this directory is a Git repo. Nothing about the directory itself has changed; Git just now knows to track it for version control.

```
ls -a
```

```
## .
## ..
## .git
```

Add files (`git add`)

Create a `README.md` using a text editor of your choice and type some text into the file. In this example, I will use the command line to create this file but feel free to use another program if you wish.

```
echo 'Hello!' > README.md
```

We see the new file in the TestRepo

```
ls -a
```

```
## .
## ..
## .git
## README.md
```

Now that we've added a file, we can query if there are any differences between our local `TestRepo` directory and the version controlled repo with

```
git status
```

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

We see that `README.md` simply being in a repo does not automatically tell Git to track this file; it remains "untracked". So, we must add the file to Git tracking. We can either add just this file specifically with

```
git add README.md
```

Or if we had more than 1 file to add at once, we could add all with

```
git add .
```

Now we see that our README is tracked but not committed.

```
git status
```

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

Git staging area (index)

If the file is tracked but not committed (*i.e.* version saved), where is it? Well, Git has a staging area (also called the index) where a version of a file is stored prior to it being more permanently saved as a committed version.

This is useful because sometimes you may want to:

- save a version as a back-up in the short-term but do not want to permanently save that version in the long-term
- complete many small changes separately but save them all together as 1 new version
- compare your current version with the last committed version without causing any conflicts in your repo

Commit changes (`git commit`)

Once you have a version you want to save for all time, you need to `commit` it from the index to the repo. It is best practices to include a message with each commit using `git commit -m "Your message here"`. You wouldn't believe the number of commits with a message simply of "typo"! We will use the following to commit our first version of `TestRepo` with the README.

```
git commit -m "Initialize README"
```

```
## [master (root-commit) 02fb85f] Initialize README
##  1 file changed, 1 insertion(+)
##  create mode 100644 README.md
```

You receive a summary of how many files were changed and what those changes where (though exact ID numbers will be specific to your repo).

Also, note the message that your name and email were automatically set based on your computer's information. In the GitHub tutorial, we will go over how to setup your computer to link to your GitHub account instead.

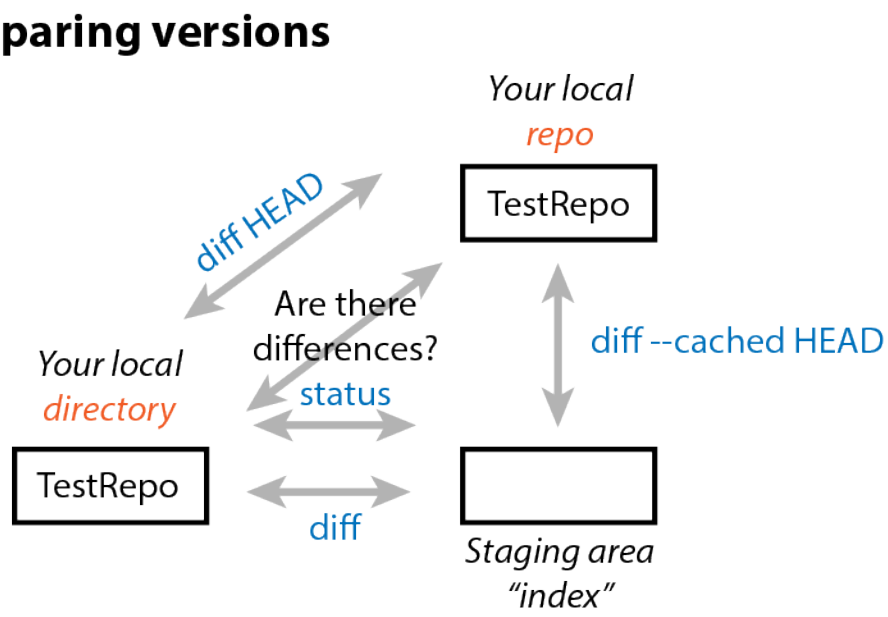
Now, we see that our local directory, index, and repo are all identical.

```
git status
```

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

Compare versions (`git status` and `diff`)

Since we are working with a single, small file in a small repo, it is easy to remember what versions are where. However, as you make larger files and repos, sometimes you will not know this. So, Git has functions for comparing versions.



Your `git status` should be up-to-date after the previous section. Now, let's create some differing versions in our local, index, and repo.

Add your birthday to `README.md` and `git add .` this change *but do not commit*.

```
echo 'My birthday is November 29.' >> README.md
git add .
```

Then add your favorite color and *do not add or commit* this change.

```
echo 'My favorite color is purple.' >> README.md
```

So, now we have versions the following versions.

- local: hello, birthday, color
- index: hello, birthday
- repo: hello

We can determine what version is where with various parameters in `git diff`. Added lines are indicated by +, deleted lines by -, and changed lines by showing both versions inline.

To compare the local and index

```
git diff
```

```
## diff --git a/README.md b/README.md
## index 016b9ef..049fca5 100644
## --- a/README.md
## +++ b/README.md
## @@ -1,2 +1,3 @@
##  Hello!
##  My birthday is November 29.
## +My favorite color is purple.
```

To compare the index and repo

```
git diff --cached HEAD
```

```
## diff --git a/README.md b/README.md
## index 10ddd6d..016b9ef 100644
## --- a/README.md
## +++ b/README.md
## @@ -1 +1,2 @@
##  Hello!
## +My birthday is November 29.
```

To compare the local and repo

```
git diff HEAD
```

```
## diff --git a/README.md b/README.md
## index 10ddd6d..049fca5 100644
## --- a/README.md
## +++ b/README.md
## @@ -1 +1,3 @@
##  Hello!
## +My birthday is November 29.
## +My favorite color is purple.
```

And so if you are unsure which version you are using (local) or which one you want to revert to, `diff` can show you.

Past commits (`git log` and `show`)

You'll notice in the `diff` commands that reference the repo, it is referred to as HEAD. This is the most recent commit version in the repo and you can go back farther with `HEAD~##`.

Let's `add and commit` our current changes so that we have multiple commits to look at. Since we never committed the version with only the birthday added, this version is lost with a new `git add`.

```
git add .
git commit -m "Add birthday and color"
```

```
## [master 26f2a77] Add birthday and color
##  1 file changed, 2 insertions(+)
```

We can see all past commits in this repo with

```
git log
```

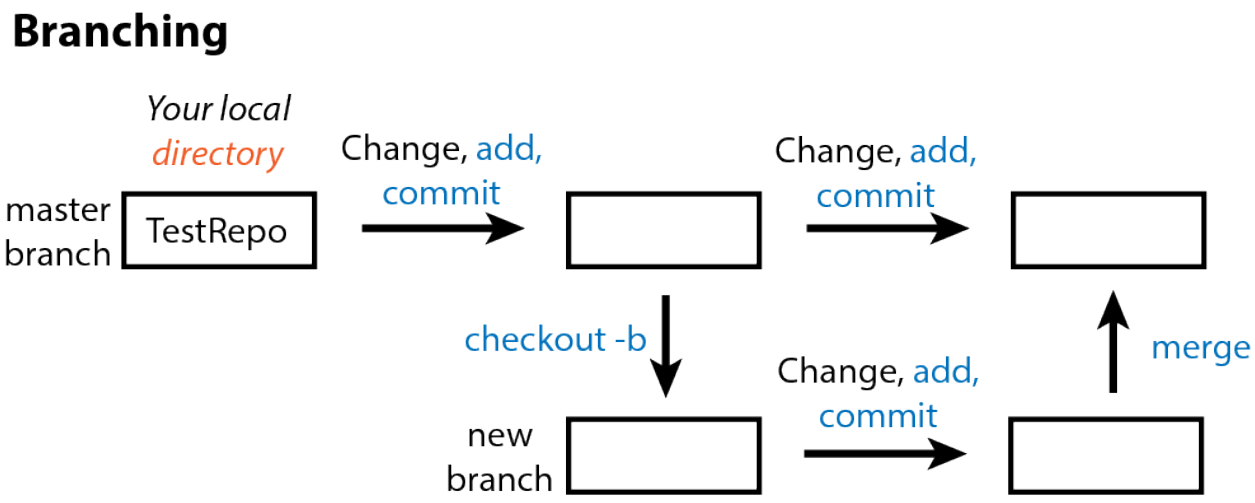
```
## commit 26f2a77cd31a281cb51e4c049a42bbcb618a12a9
## Author: Kim Dill-McFarland <kdillmcfarland@gmail.com>
## Date:   Tue Apr 2 23:04:58 2019 -0700
##
##     Add birthday and color
##
## commit 02fb85faa235744aedbed70be7ac096d0b981d4f
## Author: Kim Dill-McFarland <kdillmcfarland@gmail.com>
## Date:   Tue Apr 2 23:04:58 2019 -0700
##
##     Initialize README
```

Or view a specific past version using `HEAD~##`. For example, we can view our first commit which is `HEAD~1`

```
git show HEAD~1
```

```
## commit 02fb85faa235744aedbed70be7ac096d0b981d4f
## Author: Kim Dill-McFarland <kdillmcfarland@gmail.com>
## Date:   Tue Apr 2 23:04:58 2019 -0700
##
##     Initialize README
##
## diff --git a/README.md b/README.md
## new file mode 100644
## index 0000000..10ddd6d
## --- /dev/null
## +++ b/README.md
## @@ -0,0 +1 @@
## +Hello!
```

Branches (`git checkout -b`)



One of the best ways to avoid the need to undo anything in Git is to use branches. These are copies of your repo that you can work on independently of the "master" without impacting the master but still version control changes as you go. You can then choose to merge the branch with the master if you want to keep those changes or abandon the branch and go back the to the master if you do not want to keep those changes.

You may have noticed that many of our previous Git outputs start with `On branch master`. This is because by default, we were working on the master branch of the project (the only branch up to this point).

Branches are particularly useful to:

- test new features without breaking the current working code
- collaborate in parallel
- follow a risky idea that has the potential to break your working code
- develop the same base code into 2+ other applications

To start a new branch, first make sure that everything is up-to-date with `git status`. Then, `checkout` (*i.e.* create) and move into a new branch using the branch `-b` parameter.

```
git status
```

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

```
git checkout -b new_branch master
```

```
## Switched to a new branch 'new_branch'
```

We see that we are in the new branch.

```
git status
```

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

Now, we can make changes in this branch without impacting the master branch. Go ahead and edit `README.md` by adding another piece of information. Then `add and commit` this change in your new_branch.

```
echo 'I like cats.' >> README.md

git add .
git commit -m 'Add cats'
```

```
## [new_branch a54e2ef] Add cats
##  1 file changed, 1 insertion(+)
```

The two branches no longer match so we must merge them and decide which version(s) to keep. To do so, we must first move back into the master branch

```
git checkout master
```

```
## Switched to branch 'master'
```

Then merge the branches, which overwrites the older master branch with our new changes in the new_branch

```
git merge new_branch
```

```
## Updating 26f2a77..a54e2ef
## Fast-forward
##  README.md | 1 +
##  1 file changed, 1 insertion(+)
```

So, the README file now contains all the changes that were made in new_branch.

```
head README.md
```

```
## Hello!
## My birthday is November 29.
## My favorite color is purple.
## I like cats.
```

Merge conflicts (`git merge`)

Merging works simply if only 1 of the branches being merged has be altered since it branched off. If both branches contain non-synonymous changes, there will be a conflict.

You resolve a conflicts by merging, opening the file(s) with conflict(s), editing until you have the version you want, and saving the new, correct merged version with `add and commit`.

For example, we can make a new_branch2

```
git checkout -b new_branch2 master
```

```
## Switched to a new branch 'new_branch2'
```

Modify the README; add and commit those changes to **this branch**

```
echo 'I like dogs too!' >> README.md
git add .
git commit -m "Add dogs"
```

```
## [new_branch2 c66e921] Add dogs
##  1 file changed, 1 insertion(+)
```

Also modify the README in our master branch.

```
git checkout master

echo 'I do not like dogs.' >> README.md
git add .
git commit -m "Add dogs again"
```

```
## Switched to branch 'master'
## [master e04d8f8] Add dogs again
##  1 file changed, 1 insertion(+)
```

And now the version in the master has non-mergable changes compared to the new_branch2 version *e.g.* I can't both like and not like dogs. Thus, when we try to merge these branches, there is a conflict.

```
git merge new_branch2
```

```
## Auto-merging README.md
## CONFLICT (content): Merge conflict in README.md
## Automatic merge failed; fix conflicts and then commit the result.
```

If we open the README, Git has marked the conflict(s) like so

```
<<<<<< HEAD
master version
=======
branch version
>>>>>> name_of_branch
```

```
head README.md
```

```
## Hello!
## My birthday is November 29.
## My favorite color is purple.
## I like cats.
## <<<<<<< HEAD
## I do not like dogs.
## =======
## I like dogs too!
## >>>>>>> new_branch2
```

To resolve this, we must open the README and correct the conflict, being sure to remove the `>`, `=`, and `<` markers. Once we have the corrected version like so

```
head README.md
```

```
## Hello!
## My birthday is November 29.
## My favorite color is purple.
## I like cats.
```

```
## I like dogs too!
```

we can try adding and committing again. If we've fully resolved the conflict, there will be no errors!

```
git add .
git commit -m "Resolve dog conflict"
```

```
## [master 994d65f] Resolve dog conflict
```

And we can see our entire history, including the branches, in the log.

```
git log
```

```
## commit 994d65f2e9e9967949d3cdaeb46e44cface914a3
## Merge: e04d8f8 c66e921
## Author: Kim Dill-McFarland <kdillmcfarland@gmail.com>
## Date:   Tue Apr 2 23:04:58 2019 -0700
##
##     Resolve dog conflict
##
## commit e04d8f826f69e42afb40d6144a0114a70461b852
## Author: Kim Dill-McFarland <kdillmcfarland@gmail.com>
## Date:   Tue Apr 2 23:04:58 2019 -0700
##
##     Add dogs again
##
## commit c66e921e9b41e196b01258791583020a8a668a2e
## Author: Kim Dill-McFarland <kdillmcfarland@gmail.com>
## Date:   Tue Apr 2 23:04:58 2019 -0700
##
##     Add dogs
##
## commit a54e2ef86c19b5149b54279438870f59ec8b7f05
## Author: Kim Dill-McFarland <kdillmcfarland@gmail.com>
## Date:   Tue Apr 2 23:04:58 2019 -0700
##
##     Add cats
##
## commit 26f2a77cd31a281cb51e4c049a42bbcb618a12a9
## Author: Kim Dill-McFarland <kdillmcfarland@gmail.com>
## Date:   Tue Apr 2 23:04:58 2019 -0700
##
##     Add birthday and color
##
## commit 02fb85faa235744aedbed70be7ac096d0b981d4f
## Author: Kim Dill-McFarland <kdillmcfarland@gmail.com>
## Date:   Tue Apr 2 23:04:58 2019 -0700
##
##     Initialize README
```

Summary of Git

Thus far, we have covered the main Git terms and functions that you will need to enact version control on your own computer (summarized below).

Terms

- **Repository**: the contents (directory/folder) of a project including all history and versions for all files in that project
- **Commit**: a snapshot of the state of the repository at a given point in time including the author, time, and a comment
- **Staging/Index**: files marked for inclusion in the next commit as whatever version that they were *at the time they were staged*, not necessarily the most up-to-date version on your computer
- **Branch**: a named sequence of commits kept separate from the original "master" sequence
- **Merge**: including changes from one branch into another
- **Conflict**: parallel changes to the same section(s) of a file that can't be automatically merged

Commands

- `status`: query for differences in versions in local, index, or repo
- `init`: initialize a repo
- `add`: add files to the index (staging) prior to committing
- `commit -m`: save changes in the index as a time-stamped version with comment (message)
- `diff (HEAD) (--cached HEAD)`: compare file versions in the local vs. index (local vs. repo) (index vs. repo)
- `log`: show list of previous commits in the current repo
- `show`: show a detail account of a previous commit
- `checkout`: create and move between branches or the local, index, and repo
  - Not shown here but `checkout HEAD` can be used to move to a past commit and re-instate file versions from that commit
- `merge`: combine changes from a branch into the master repo (may result in conflicts)

To checkout more functionalities in Git, see a list of all Git commands.
