

# Git Walkthrough

This is a script of sorts for running the Version Control with Git portion of a Software Carpentry Workshop for Digital Humanities researchers. It is based on content from the Software Carpentry git-novice lessons. It diverges from these lessons by placing an emphasis on using GitHub that is not part of the original lessons, particularly using GitHub and markdown to create a simple project website to share work.

## Why Git/GitHub?

---

There are two reasons to use Git/GitHub:

1. Version control
2. Sharing

### Why (automated) version control

The importance of version control can begin to be understood with the following comic from Jorge Cham's ["Piled Higher and Deeper"](#)

# "FINAL".doc



FINAL.doc!



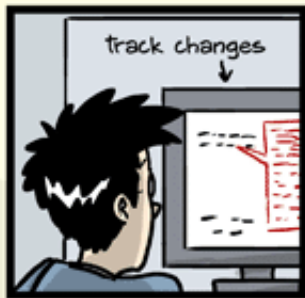
FINAL\_rev.2.doc



FINAL\_rev.6.COMMENTS.doc



FINAL\_rev.8.comments5.  
CORRECTIONS.doc



FINAL\_rev.18.comments7.  
corrections9.MORE.30.doc



FINAL\_rev.22.comments49.  
corrections.10.##\$%WHYDID  
ICOMETOGRADSCHOOL?????.doc



JORGE CHAM © 2012

WWW.PHDCOMICS.COM

Rather than keep multiple slight variations of a document through its entire life cycle version control allows for *one* copy of a document to be kept, the most recent, while all the changes made to get it to its current state are held in the

background by a program specially designed to track this information. This saves the need for bizzare naming conventions and holding multiple copies of the same document.

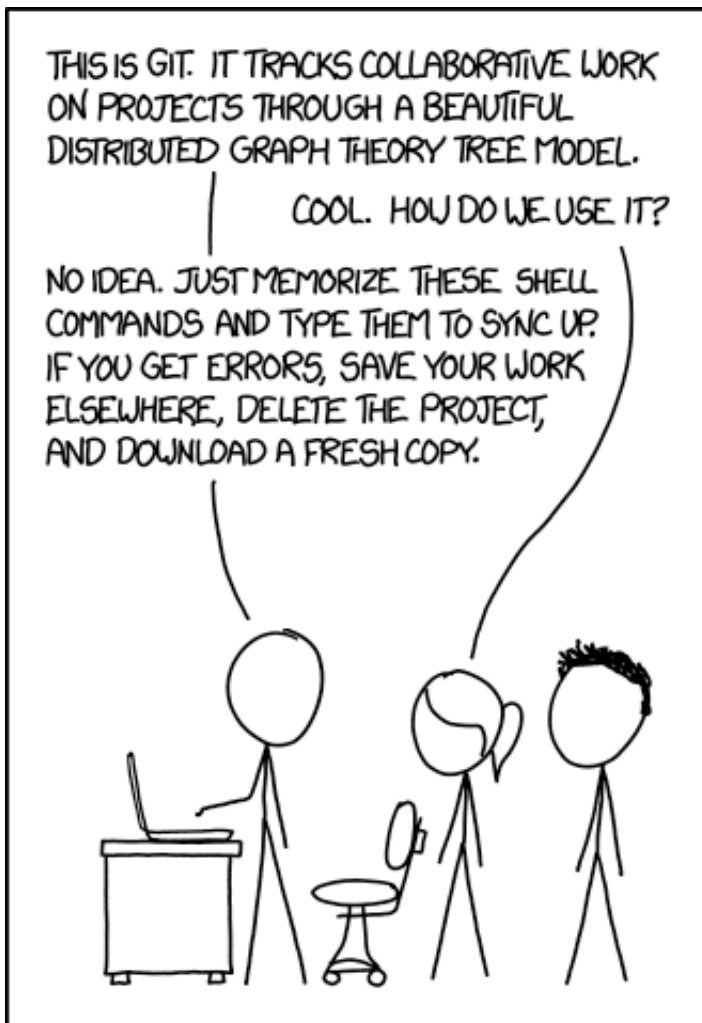
Version control goes well beyond avoiding bizarre naming conventions though and really comes into its own when there are multiple authors working on a document in parallel. Rather than just accept what any other contributor has written, version control allows those responsible for the document to choose exactly what gets kept and what gets changed. It even allows for the roll back of content to previous versions when it is determined that those versions are preferred. Combined with a central repository like GitHub it also provides a backup for your data and the benefits of enterprise class redundant systems.

## **Sharing**

While sharing information more easily was part of the original purposes for developing Git the ease with which this can be done via GitHub has made it the case that many users do very little with version control at all and use Git/GitHub solely for it's ability to quickly produce web sites for the sharing of information. GitHub facilitates the Open science Model by giving researchers a publishing platform that provides access to data, papers, presentations, and the like, all through a fairly easy to use web interface.

## **The problem with Git**

---



From <https://xkcd.com/1597/> by Randal Munroe

## Learning Objectives

---

We're going to make sharing information via GitHub the primary objective of this portion of the workshop and look at version control only in light of sharing. By doing this we will be placing the emphasis on using GitHub rather than on using Git. Make no mistake though Git is central to GitHub and must be understood to make effective use of it. There will be no commandline work in this section but the command line git tutorial is available to anyone who wants it. Emphasis will be placed on generating an understanding of the core concepts that power Git and the terms used to refer to these concepts.

GitHub now has a really excellent series of guides/tutorials that can be found at <https://guides.github.com/>.

## "Hello World!" Tutorial

---

GitHub now offers a short tutorial that will give you the core concepts of the system without having to "know how to code, use the command line, or install Git (the version control software that GitHub is built on)." We'll work through this tutorial as a class and then build on this by installing Git, using the command line, and expanding on the website capabilities of GitHub.

## Get an account

Before we can use the tutorial you will need to have a GitHub account. You can get this at [github.com](https://github.com). Visit the site, click the "Sign up" button at the top of the page, and follow the process.

Once this is done we will begin following the tutorial that can be found at <https://guides.github.com/activities/hello-world/>.


What follows for the rest of this section is mostly extraced verbatim from the tutorial. Content not from the tutorial is prefaced with "!!!".

## Create a repository

1. In the upper right corner, next to your username, click **+** and then click **New repository**.
2. Name your repository 'hello-world'.
3. Write a short description.
4. !!! Leave the repository as **Public**. Public repositories are free on GitHub and are open to the world. Private repositories typically have a monthly fee associated with them (as little as \$7 for up to 5 repositories). Exceptions are made for educational related work such as teaching. See [https://education.github.com/guide/private\\_repos](https://education.github.com/guide/private_repos)
5. Select **Initialize this repository with a README**.
6. !!! Ignore the .gitignore and license options. .gitignore would hold a list of files to not track, such as transient/temporary files created by various programs while editing code and the like which are only useful to that program at that moment.
7. Click **Create repository**.

PUBLIC

Owner

 hubot


Repository name

hello-world


Great repository names are short and memorable. Need inspiration? How about **petulant-shame**.

Description (optional)

Just another repository

☒  **Public**

Anyone can see this repository. You choose who can commit.

☐  **Private**


You choose who can see and commit to this repository.

☒ **Initialize this repository with a README**

This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.

Add .gitignore: **None**

Add a license: **None**



Create repository

## Explaining branching

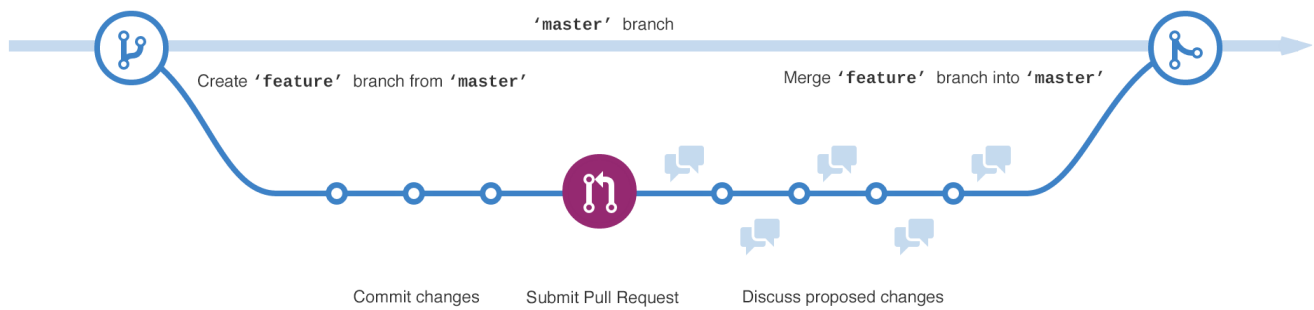
**Branching** is the way to work on different versions of a repository at one time.

By default your repository has one branch named `master` which is considered to be the definitive branch. We use branches to experiment and make edits before committing them to `master`.

When you create a branch off the `master` branch, you're making a copy, or snapshot, of `master` as it was at that point in time. If someone else made changes to the `master` branch while you were working on your branch, you could pull in those updates.

This diagram shows:

- The `master` branch
- A new branch called `feature` (because we're doing 'feature work' on this branch)
- The journey that `feature` takes before it's merged into `master`.
- `feature` merging back into `master`.



Have you ever saved different versions of a file? Something like:

- `story.txt`
- `story-joe-edit.txt`
- `story-joe-edit-reviewed.txt`

Branches accomplish similar goals in GitHub repositories.


Here at GitHub, our developers, writers, and designers use branches for keeping bug fixes and feature work separate from our `master` (production) branch. When a change is ready, they merge their branch into `master`.


A straightforward walkthrough of branching as part of the GitHub workflow can be found at <https://guides.github.com/introduction/flow/>. It is recommended that this is reviewed.


## Creating a new branch


1. Go to your new repository `hello-world`.
2. Click the drop down at the top of the file list that says **branch: master**.
3. Type a branch name, `readme-edits`, into the new branch text box.
4. Select the blue **Create branch** box or hit “Enter” on your keyboard.

## Just another repository — Edit


 **1** commit


 **1** branch


 branch: **master** ▾

**hello-world** / 

Initial commit

 **hubot** authored just now

 **README.md** Initial

 **README.md**

Now you have two branches, `master` and `readme-edits`. They look exactly the same, but not for long! Next we'll add our changes to the new branch.

### Make and commit changes

Bravo! Now, you're on the code view for your readme-edits branch, which is a copy of master. Let's make some edits.

On GitHub, saved changes are called commits. Each commit has an associated commit message, which is a description explaining why a particular change was made. Commit messages capture the history of your changes, so other contributors can understand what you've done and why.

#### Make and commit changes

1. Click the `README.md` file.
2. Click the pencil icon in the upper right corner of the file view to edit.
3. In the editor, write a bit about yourself.
4. Write a commit message that describes your changes (at the bottom of the page).
5. Ignore the option to "Create a new branch and start a pull request". We'll look at what this means and how to do it later.



6. Click the **Commit changes** button.

The screenshot shows the GitHub web interface for the repository 'hubot / hello-world'. At the top, there are navigation links for Code, Issues (0), Pull requests (0), Wiki, Pulse, Graphs, and Settings. Below the repository name, the file 'README.md' is selected for editing. The editor shows the following content:

```
1 # hello-world
2
3 Hi Humans!
4
5 Hubot here, I like Node.js and Coffeescript (that's what I'm made of!).
6 I've had tacos on the moon and find them far superior to Earth tacos.
7
```

Below the editor is the 'Commit changes' dialog. It has a title bar with 'Commit changes' and a 'Preview changes' button. The dialog contains a text input field with 'Finish README' and a larger text area with 'And mention moon tacos'. There are two radio buttons for the commit target:

- ☒ Commit directly to the `readme-edits` branch
- ☐ Create a new branch for this commit and start a pull request. [Learn more about pull requests.](#)

At the bottom of the dialog are two buttons: 'Commit changes' (green) and 'Cancel' (grey).

These changes will be made to just the README file on your `readme-edits` branch, so now this branch contains content that's different from `master`.

Note that commits are **manual**. You must explicitly make them. Commit and commit regularly.

## Open a pull request

Nice edits! Now that you have changes in a branch off of `master`, you can open a *pull request*.

Pull Requests are the heart of collaboration on GitHub and amount to asking someone to merge your content with theirs. When you open a *pull request*, you're proposing your changes and requesting that someone review and pull in your contributions into their branch. Pull requests show the differences, or *diffs*, of the content from both branches. The changes, additions, and subtractions are shown in green and red.

As soon as you make a commit, you can open a pull request and start a discussion, even before the code is finished.

By using GitHub's @mention system in your pull request message, you can ask for feedback from specific people or teams, whether they're down the hall or 10 time zones away.

You can even open pull requests in your own repository and merge them yourself. It's a great way to learn the GitHub Flow before working on larger projects.

Step I Screenshot -I-: Click the **Pull Request** tab, then from the Pull Request page, click the green **New pull request** button. I ? Select the branch you made, `readme-edits`, to compare with `master` (the original). I ? Look over your changes in the diffs on the Compare page, make sure they're what you want to submit. I ? When you're satisfied that these are the changes you want to submit, click the big green **Create Pull Request** button. I ? Give your pull request a title and write a brief description of your changes. I ?

When you're done with your message, click **Create pull request!**

## Merge your pull request

In this final step, it's time to bring your changes together – merging your readme-edits branch into the master branch.

1. Click the green Merge pull request button to merge the changes into master.
2. Click Confirm merge.
3. Go ahead and delete the branch, since its changes have been incorporated, with the Delete branch button in the purple box.



**This branch has no conflicts with the base branch**  
Merging can be performed automatically.



**Merge pull request**

You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

look into creating a conflict

## Reverting pull request

Reverting a pull request on GitHub creates a new pull request that contains one revert of the merge commit from the original merged pull request.

Note: You may need to [use Git to manually revert the individual commits](#) if:

- Reverting the pull request causes merge conflicts

- The original pull request was not originally merged on GitHub (for example, using a fast-forward merge on the command line)

1. Under your repository name, click Pull requests.
2. Click on **Closed** to see a list of past pull requests that have been merged.
3. In the "Pull Requests" list, click the pull request you'd like to revert. In this case it will be the request that we just accepted.
4. Near the bottom of the pull request, click **Revert**.
5. Merge the resulting pull request.
6. Delete the branch made that created this delete.

## Linking your desktop

---

GitHub is a nice place to start but it makes a lot of assumptions that may not reflect reality, such as:

1. You always have an Internet connection when you want to work on things.
2. You only have text files.
3. You are happy doing all your edits in the GitHub editor.
4. You are happy with the limited control that GitHub gives you over version control via its interface.

Sooner or later one of these assumptions is likely going to be false and when this happens you should look at having Git run on your desktop. This can be done either via a desktop application or via the commandline. Note that the commandline tool is *much* more powerful than the desktop app. The commandline tool is also potentially *much* more confusing and since the point of this portion of the workshop is to get you comfortable using Git for sharing and non-complex version control we'll look at the desktop application now and may return to the commandline tool at the end.

**Warning:** The desktop app is only available for Windows and Mac. Linux users will be stuck with the commandline.

## Getting set up

1. The commandline tool can be downloaded for either Mac or Windows at: <https://desktop.github.com/>
2. Installation is straightforward. You will be asked for your GitHub account name and password and then you will be presented with the core interface.
3. Initially this will be empty (unless you already have Git repositories on your computer) so we will need to add your hello-world project. Do this by clicking the **+** in the top left corner of the screen, clicking **clone** and then selecting "hello-world". (A clone is an exact copy of a repository that you own.)
4. It will ask you where to put this directory. Put it on the desktop so that it can be easily found.

Go through the parts of the window with them and show how they could revert to an earlier work by clicking at a point on the tree and choosing the **Revert** option from the gear/settings menu.

## Pushing changes directly to master from the desktop

1. Navigate to the hello-world folder on the Desktop and open `README.md` with a text editing tool. Let's build on the work from earlier this morning and use `nano` if possible.
2. Add some new text to the file.
3. Change a period earlier in the file to an exclamation point.
4. Return to the desktop app and notice that the top of the screen now says that there is **1 Uncommitted Change**. Click on this notice to review the change.
5. Clicking on the blue bars on the right will turn the associated change on and off. If you turn off one part of a change make sure to turn off the accompanying part.
6. Add a summary and description.
7. Click **sync**.
8. View the results on GitHub.

**Challenge:** Discover what happens when the deletion of a line is uncoupled from its replacement (and vice versa) when making commits and then syncing with the repository. **Result:** If you only commit the deletion then the line will be deleted and the replacement will not be added. If you only commit the addition then the original line will be left and the addition appended to it. Consequently it is best **not** to uncouple changes since the results will typically end in confusion.

## Branching on the desktop

Branching is done on the desktop by selecting the right repository and clicking the branch button.

It is important to watch which branch is active in the tree-view.

When done click the **pull request** button.

Branches can be deleted by having the branch active and then choosing **Delete ...** from the **Branch** menu.

<https://guides.github.com/introduction/getting-your-project-on-github/> <https://help.github.com/desktop/guides/>  
<https://help.github.com/desktop/guides/contributing/reverting-a-commit/>

## Adding a project

This is as easy as dragging a folder onto the app and then synchronizing it.

You can also click the **+** in the top left corner, then click the "Add" tab, and follow along from there.

# Markdown

---

Use an online editor to make the rendering of markdown realtime.

- <https://stackedit.io/>
- <http://dillinger.io/>

Core things to show:

- #s to make headings
- \*s to make italics and bold
- \*s and \1s to make lists
- `s and ""s and indents to make code blocks
- >s to make quotes
- []() to make links
- ![]() to add images
- \ to escape special characters
- RAW button on GitHub md / code pages

Have them make some changes to their README.md and commit them.

Point them to some online guide, perhaps <https://guides.github.com/features/mastering-markdown/>

## Other benefits

---

GitHub provides other benefits over the ones looked at so far in detail. These are beyond the scope of this introduction but you can get a sense for what is possible by looking at the GitHub guides found at: <https://guides.github.com>. The most relevant to the work of participants are highlighted here.

## Collaborating

Giving people access to controlling repositories. Doesn't seem to be a GitHub tutorial on this.

## Issues

<https://guides.github.com/features/issues/> Have participants share their hello-world repositories on the Etherpad and partner with each other to push an issue or two.

## Forking

<https://guides.github.com/activities/forking> Must be done from either GitHub or from the commandline.

## Web pages

<https://guides.github.com/features/pages> We won't go here in this workshop since it opens up the necessity of HTML syntax. We'll just stick with the markdown based README.md page for the time being and anyone can follow along with the tutorial later if they would like.

Could show off [Jentery's page](#) which is a nice slide show/presentation just to illustrate what is possible.

## Citations

<https://guides.github.com/activities/citable-code>

# Commandline control

---

## setting up Git

!!! What happens in Git Desktop when each of these sections is carried out???

Git is a program just like the programs invoked by `ls`, `mv`, and the other commands we looked at during the Commandline portion of this class. The command to run Git is just `git`.

```
$ git
```

Given the range of things that Git is able to do just running `git` isn't enough information and you'll be presented with a list of possible commands. `git` needs to be followed by a term drawn from a special vocabulary of verbs/actions and often each of these needs to be followed by its own set of vocabulary/flags. We'll only look at the basics of these so that you begin to get comfortable with Git on the commandline and see that it is the commandline tool that is really running everything behind the scenes in both GitHub and Git Desktop.

The first thing that we will set are four global variables, as follows:

```
$ git config --global user.name "Vlad Dracula"  
$ git config --global user.email "vlad@tran.sylvan.ia"  
$ git config --global color.ui "auto"  
$ git config --global core.editor "nano -w"
```

These first two variables set the ID information that your changes will be signed with. The third ensures that

your colour scheme will match the one you see on the screen for the rest of this section. The fourth is how to setup the default text editor if changes need to be made. Git often defaults to a text editor called Vim, which, although having many laudable features and a bit of a cult following among programmers, has a hidden and non-intuitive interface and so it is often best to swap it for something friendly. We've chosen **nano** here but if someone really wants an alternative then the following table may satisfy them.

Editor	Configuration command
Text Wrangler	\$ git config --global core.editor "edit -w"
Sublime Text (Mac)	\$ git config --global core.editor "subl -n -w"
Sublime Text (Win)	\$ git config --global core.editor "'c:/program files/sublime text 2/sublime_text.exe' -w"
Notepad++ (Win)	\$ git config --global core.editor "'c:/program files (x86)/Notepad++/notepad++.exe' - multiInst -notabbar -nosession -noPlugin"
Kate (Linux)	\$ git config --global core.editor "kate"
Gedit (Linux)	\$ git config --global core.editor "gedit -s"

If you ever want to check your core settings---or to see what all the available settings are---you can use:

```
$ git config --list
```

## Creating a repository

Once Git is configured, we can start using it. Let's create a directory for our work and then move into that directory:

```
$ mkdir planets
$ cd planets
```

Then we tell Git to make `planets` a [repository](#)—a place where Git can store versions of our files:

```
$ git init
```

If we use `ls` to show the directory's contents, it appears that nothing has changed:

```
$ ls
```

But if we add the `-a` flag to show everything, we can see that Git has created a hidden directory within `planets` called `.git`:

```
$ ls -a  
  
.  ..  .git
```

Git stores information about the project in this special sub-directory. If we ever delete it, we will lose the project's history.

We can check that everything is set up correctly by asking Git to tell us the status of our project:

```
$ git status  
  
# On branch master  
#  
# Initial commit  
#  
nothing to commit (create/copy files and use "git add" to track)
```

### Places to Create Git Repositories {.challenge}

Dracula starts a new project, `moons`, related to his `planets` project. Despite Wolfman's concerns, he enters the following sequence of commands to create one Git repository inside another:

```
cd                # return to home directory  
mkdir planets    # make a new directory planets  
cd planets        # go into planets  
git init          # make the planets directory a Git repository  
mkdir moons      # make a sub-directory planets/moons  
cd moons          # go into planets/moons  
git init          # make the moons sub-directory a Git repository
```

Why is it a bad idea to do this? How can Dracula "undo" his last `git init` ?

The problem is that there is now a Git repository inside a Git repository making it the case that the secondary repository is now being tracked by the first. Knowing that *all* the information about a repository is tracked in a hidden file called `.git` inside any repository Dracula can just delete this folder and all its content.

## Tracking Changes



Let's create a file called `mars.txt` that contains some notes about the Red Planet's suitability as a base. (We'll use `nano` to edit the file; you can use whatever editor you like. In particular, this does not have to be the `core.editor` you set globally earlier.)

```
$ nano mars.txt
```

Type the text below into the `mars.txt` file:

```
Cold and dry, but everything is my favorite color
```

`mars.txt` now contains a single line, which we can see by running:

```
$ ls

mars.txt

$ cat mars.txt

Cold and dry, but everything is my favorite color
```

If we check the status of our project again, Git tells us that it's noticed the new file:

```
$ git status

# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   mars.txt
nothing added to commit but untracked files present (use "git add" to track)
```

The "untracked files" message means that there's a file in the directory that Git isn't keeping track of. We can tell Git to track a file using `git add`:

```
$ git add mars.txt
```

and then check that the right thing happened:

```
$ git status

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

Git now knows that it's supposed to keep track of `mars.txt`, but it hasn't recorded these changes as a commit yet. To get it to do that, we need to run one more command:

```
$ git commit -m "Start notes on Mars as a base"

[master (root-commit) f22b25e] Start notes on Mars as a base
1 file changed, 1 insertion(+)
create mode 100644 mars.txt
```

When we run `git commit`, Git takes everything we have told it to save by using `git add` and stores a copy permanently inside the special `.git` directory. This permanent copy is called a [commit](#) (or [revision](#)) and its short identifier is `f22b25e` (Your commit may have another identifier.)

We use the `-m` flag (for "message") to record a short, descriptive, and specific comment that will help us remember later on what we did and why. If we just run `git commit` without the `-m` option, Git will launch `nano` (or whatever other editor we configured as `core.editor`) so that we can write a longer message.

[Good commit messages][commit-messages] start with a brief (<50 characters) summary of changes made in the commit. If you want to go into more detail, add a blank line between the summary line and your additional notes.

If we run `git status` now:

```
$ git status

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

it tells us everything is up to date. If we want to know what we've done recently, we can ask Git to show us the project's history using `git log`:

```
$ git log

commit f22b25e3233b4645dabd0d81e651fe074bd8e73b
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 09:51:46 2013 -0400

Start notes on Mars as a base
```

`git log` lists all commits made to a repository in reverse chronological order. The listing for each commit includes the commit's full identifier (which starts with the same characters as the short identifier printed by the `git commit` command earlier), the commit's author, when it was created, and the log message Git was given when the commit was created.

### Where Are My Changes? {.callout}

If we run `ls` at this point, we will still see just one file called `mars.txt`. That's because Git saves information about files' history in the special `.git` directory mentioned earlier so that our filesystem doesn't become cluttered (and so that we can't accidentally edit or delete an old version).

Now suppose Dracula adds more information to the file. (Again, we'll edit with `nano` and then `cat` the file to show its contents; you may use a different editor, and don't need to `cat .`)

```
$ nano mars.txt
$ cat mars.txt

Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
```

When we run `git status` now, it tells us that a file it already knows about has been modified:

```
$ 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:   mars.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

The last line is the key phrase: "no changes added to commit". We have changed this file, but we haven't told

Git we will want to save those changes (which we do with `git add`) nor have we saved them (which we do with `git commit`). So let's do that now. It is good practice to always review our changes before saving them. We do this using `git diff`. This shows us the differences between the current state of the file and the most recently saved version:

```
$ git diff

diff --git a/mars.txt b/mars.txt
index df0654a..315bf3a 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,2 @@
Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
```

The output is cryptic because it is actually a series of commands for tools like editors and `patch` telling them how to reconstruct one file given the other. If we break it down into pieces:

1. The first line tells us that Git is producing output similar to the Unix `diff` command comparing the old and new versions of the file.
2. The second line tells exactly which versions of the file Git is comparing; `df0654a` and `315bf3a` are unique computer-generated labels for those versions.
3. The third and fourth lines once again show the name of the file being changed.
4. The remaining lines are the most interesting, they show us the actual differences and the lines on which they occur. In particular, the `+` markers in the first column show where we have added lines.

After reviewing our change, it's time to commit it:

```
$ git commit -m "Add concerns about effects of Mars' moons on Wolfman"
$ 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:   mars.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Whoops: Git won't commit because we didn't use `git add` first. Let's fix that:

```
$ git add mars.txt
$ git commit -m "Add concerns about effects of Mars' moons on Wolfman"

[master 34961b1] Add concerns about effects of Mars' moons on Wolfman
1 file changed, 1 insertion(+)
```

Git insists that we add files to the set we want to commit before actually committing anything because we may not want to commit everything at once. For example, suppose we're adding a few citations to our supervisor's work to our thesis. We might want to commit those additions, and the corresponding addition to the bibliography, but *not* commit the work we're doing on the conclusion (which we haven't finished yet).

To allow for this, Git has a special *staging area* where it keeps track of things that have been added to the current [change set](#) but not yet committed.

**Staging area {.callout}** If you think of Git as taking snapshots of changes over the life of a project, `git add` specifies *what* will go in a snapshot (putting things in the staging area), and `git commit` then *actually takes* the snapshot, and makes a permanent record of it (as a commit). If you don't have anything staged when you type `git commit`, Git will prompt you to use `git commit -a` or `git commit --all`, which is kind of like gathering *everyone* for the picture! However, it's almost always better to explicitly add things to the staging area, because you might commit changes you forgot you made. (Going back to snapshots, you might get the extra with incomplete makeup walking on the stage for the snapshot because you used `-a`!) Try to stage things manually, or you might find yourself searching for "git undo commit" more than you would like!



Let's watch as our changes to a file move from our editor to the staging area and into long-term storage. First, we'll add another line to the file:

```
$ nano mars.txt
$ cat mars.txt
```

Cold and dry, but everything is my favorite color  
The two moons may be a problem for Wolfman  
But the Mummy will appreciate the lack of humidity

```
$ git diff

diff --git a/mars.txt b/mars.txt
index 315bf3a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,2 +1,3 @@
 Cold and dry, but everything is my favorite color
 The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

So far, so good: we've added one line to the end of the file (shown with a `+` in the first column). Now let's put that change in the staging area and see what `git diff` reports:

```
$ git add mars.txt
$ git diff
```

There is no output: as far as Git can tell, there's no difference between what it's been asked to save permanently and what's currently in the directory. However, if we do this:

```
$ git diff --staged

diff --git a/mars.txt b/mars.txt
index 315bf3a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,2 +1,3 @@
 Cold and dry, but everything is my favorite color
 The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

it shows us the difference between the last committed change and what's in the staging area. Let's save our changes:

```
$ git commit -m "Discuss concerns about Mars' climate for Mummy"

[master 005937f] Discuss concerns about Mars' climate for Mummy
1 file changed, 1 insertion(+)
```

check our status:

```
$ git status

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

and look at the history of what we've done so far:

```
$ git log

commit 005937fbe2a98fb83f0ade869025dc2636b4dad5
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date: Thu Aug 22 10:14:07 2013 -0400

    Discuss concerns about Mars' climate for Mummy

commit 34961b159c27df3b475cfe4415d94a6d1fcd064d
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date: Thu Aug 22 10:07:21 2013 -0400

    Add concerns about effects of Mars' moons on Wolfman

commit f22b25e3233b4645dabd0d81e651fe074bd8e73b
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date: Thu Aug 22 09:51:46 2013 -0400

    Start notes on Mars as a base
```

To recap, when we want to add changes to our repository, we first need to add the changed files to the staging area ( `git add` ) and then commit the staged changes to the repository ( `git commit` ):



### Choosing a commit message {.challenge}

Which of the following commit messages would be most appropriate for the last commit made to `mars.txt` ?

1. "Changes"
2. "Added line 'But the Mummy will appreciate the lack of humidity' to mars.txt"
3. "Discuss effects of Mars' climate on the Mummy"

### Committing Changes to Git {.challenge}

Which command(s) below would save the changes of `myfile.txt` to my local Git repository?

1. `$ git commit -m "my recent changes"`
2. `$ git init myfile.txt`  
`$ git commit -m "my recent changes"`
3. `$ git add myfile.txt`  
`$ git commit -m "my recent changes"`
4. `$ git commit -m myfile.txt "my recent changes"`

### **bio** Repository {.challenge}

Create a new Git repository on your computer called `bio`. Write a three-line biography for yourself in a file called `me.txt`, commit your changes, then modify one line, add a fourth line, and display the differences between its updated state and its original state.

### **Author and Committer** {.challenge}

For each of the commits you have done, Git stored your name twice. You are named as the author and as the committer. You can observe that by telling Git to show you more information about your last commits:

```
$ git log --format=full
```

When committing you can name someone else as the author:

```
$ git commit --author="Vlad Dracula vlad@tran.sylvan.ia"
```

Create a new repository and create two commits: one without the `--author` option and one by naming a colleague of yours as the author. Run `git log` and `git log --format=full`. Think about ways how that can allow you to collaborate with your colleagues.

## **Ignore files**

### **Learning Objectives** {.objectives}

---

- Configure Git to ignore specific files.
- Explain why ignoring files can be useful.

What if we have files that we do not want Git to track for us, like backup files created by our editor or intermediate files created during data analysis. Let's create a few dummy files:



```
$ mkdir results
$ touch a.dat b.dat c.dat results/a.out results/b.out
```

and see what Git says:

```
$ git status

# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   a.dat
#   b.dat
#   c.dat
#   results/
nothing added to commit but untracked files present (use "git add" to track)
```

Putting these files under version control would be a waste of disk space. What's worse, having them all listed could distract us from changes that actually matter, so let's tell Git to ignore them.

We do this by creating a file in the root directory of our project called `.gitignore`:

```
$ nano .gitignore
$ cat .gitignore

*.dat
results/
```

These patterns tell Git to ignore any file whose name ends in `.dat` and everything in the `results` directory. (If any of these files were already being tracked, Git would continue to track them.)

Once we have created this file, the output of `git status` is much cleaner:

```
$ git status
```

```
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   .gitignore
nothing added to commit but untracked files present (use "git add" to track)
```

The only thing Git notices now is the newly-created `.gitignore` file. You might think we wouldn't want to track it, but everyone we're sharing our repository with will probably want to ignore the same things that we're ignoring. Let's add and commit `.gitignore`:

```
$ git add .gitignore
$ git commit -m "Add the ignore file"
$ git status

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

As a bonus, using `.gitignore` helps us avoid accidentally adding to the repository files that we don't want to track:

```
$ git add a.dat

The following paths are ignored by one of your .gitignore files:
a.dat
Use -f if you really want to add them.
fatal: no files added
```

If we really want to override our ignore settings, we can use `git add -f` to force Git to add something. We can also always see the status of ignored files if we want:

```
$ git status --ignored

# On branch master
# Ignored files:
#   (use "git add -f <file>..." to include in what will be committed)
#
#       a.dat
#       b.dat
#       c.dat
#       results/

nothing to commit, working directory clean
```

## Ignoring nested files {.challenge}

---

Given a directory structure that looks like:

```
results/data results/plots
```

How would you ignore only `results/plots` and not `results/data` ?

## Including specific files {.challenge}

---

How would you ignore all `.data` files in your root directory except for `final.data` ? Hint: Find out what `!` (the exclamation point operator) does

## Ignoring files deep in a directory {.challenge}

---

Given a directory structure that looks like:

```
results/data/position/gps/useless.data results/plots
```

What's the shortest `.gitignore` rule you could write to ignore all `.data` files in `result/data/position/gps` Hint: What does appending `**` to a rule accomplish?

## The order of rules {.challenge}

---

Given a `.gitignore` file with the following contents:

```
.data !.data
```

What will be the result?

## Working with GitHub

### Learning Objectives {.objectives}

- Explain what remote repositories are and why they are useful.
- Clone a remote repository.
- Push to or pull from a remote repository.

Version control really comes into its own when we begin to collaborate with other people. We already have most of the machinery we need to do this; the only thing missing is to copy changes from one repository to another.

Systems like Git allow us to move work between any two repositories. In practice, though, it's easiest to use

one copy as a central hub, and to keep it on the web rather than on someone's laptop. Most programmers use hosting services like [GitHub](#), [BitBucket](#) or [GitLab](#) to hold those master copies; we'll explore the pros and cons of this in the final section of this lesson.

Let's start by sharing the changes we've made to our current project with the world. Log in to GitHub, then click on the icon in the top right corner to create a new repository called `planets` :



Name your repository "planets" and then click "Create Repository":



As soon as the repository is created, GitHub displays a page with a URL and some information on how to configure your local repository:



This effectively does the following on GitHub's servers:

```
$ mkdir planets
$ cd planets
$ git init
```

Our local repository still contains our earlier work on `mars.txt` , but the remote repository on GitHub doesn't contain any files yet:



The next step is to connect the two repositories. We do this by making the GitHub repository a [remote](#) for the local repository. The home page of the repository on GitHub includes the string we need to identify it:



Click on the 'HTTPS' link to change the [protocol](#) from SSH to HTTPS.

## HTTPS vs SSH {.callout}

We use HTTPS here because it does not require additional configuration. After the workshop you may want to set up SSH access, which is a bit more secure, by following one of the great tutorials from [GitHub](#), [Atlassian/BitBucket](#) and [GitLab](#) (this one has a screencast).



Copy that URL from the browser, go into the local `planets` repository, and run this command:

```
$ git remote add origin https://github.com/vlad/planets.git
```

Make sure to use the URL for your repository rather than Vlad's: the only difference should be your username instead of `vlad`.

We can check that the command has worked by running `git remote -v`:

```
$ git remote -v

origin    https://github.com/vlad/planets.git (push)
origin    https://github.com/vlad/planets.git (fetch)
```

The name `origin` is a local nickname for your remote repository: we could use something else if we wanted to, but `origin` is by far the most common choice.

Once the nickname `origin` is set up, this command will push the changes from our local repository to the repository on GitHub:

```
$ git push origin master

Counting objects: 9, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (9/9), 821 bytes, done.
Total 9 (delta 2), reused 0 (delta 0)
To https://github.com/vlad/planets
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

## Proxy {.callout}

If the network you are connected to uses a proxy there is an chance that your last command failed with "Could not resolve hostname" as the error message. To solve this issue you need to tell Git about the proxy:

```
$ git config --global http.proxy http://user:password@proxy.url $ git config --global https.proxy
```

`http://user:password@proxy.url`

When you connect to another network that doesn't use a proxy you will need to tell Git to disable the proxy using:

```
$ git config --global --unset http.proxy $ git config --global --unset https.proxy
```

## Password Managers {.callout}

---

If your operating system has a password manager configured, `git push` will try to use it when it needs your username and password. If you want to type your username and password at the terminal instead of using a password manager, type:

```
$ unset SSH_ASKPASS
```

You may want to add this command at the end of your `~/.bashrc` to make it the default behavior.

Our local and remote repositories are now in this state:



## The '-u' Flag {.callout}

---

You may see a `-u` option used with `git push` in some documentation. It is related to concepts we cover in our intermediate lesson, and can safely be ignored for now.

We can pull changes from the remote repository to the local one as well:

```
$ git pull origin master
```

```
From https://github.com/vlad/planets
```

```
* branch          master      -> FETCH_HEAD
```

```
Already up-to-date.
```

Pulling has no effect in this case because the two repositories are already synchronized. If someone else had pushed some changes to the repository on GitHub, though, this command would download them to our local repository.

## GitHub GUI {.challenge}

---

Browse to your `planets` repository on GitHub. Under the Code tab, find and click on the text that says "XX commits" (where "XX" is some number). Hover over, and click on, the three buttons to the right of each commit. What information can you gather/explore from these buttons? How would you get that same information in the shell?

## GitHub Timestamp {.challenge}

---

Create a remote repository on GitHub. Push the contents of your local repository to the remote. Make changes to your local repository and push these changes. Go to the repo you just created on Github and check the [timestamps](#) of the files. How does GitHub record times, and why?

## Push vs. commit {.challenge}

---

In this lesson, we introduced the "git push" command. How is "git push" different from "git commit"?

## Fixing up remote settings {.challenge}

---

It happens quite often in practice that you made a typo in the remote URL. This exercise is about how to fix this kind of issues. First start by adding a remote with an invalid URL:

```
git remote add broken https://github.com/this/url/is/invalid
```

Do you get an error when adding the remote? Can you think of a command that would make it obvious that your remote URL was not valid? Can you figure out how to fix the URL (tip: use `git remote -h`)? Don't forget to clean up and remove this remote once you are done with this exercise.

Draw from Software Carpentry. Also help available at <https://help.github.com/> , particularly Bootcamp, Setup, and Using Git.

They need to be able to:

1. create a repository and turn on Git
2. Make commits
3. Add files
4. Push to GitHub
5. Pull from GitHub
6. See integration with desktop
7. Turn Git off on a directory

8. Creating a repository Recording changes to files: add, commit, ... Viewing changes: status, diff, ... Ignoring files Working on the web: clone, pull, push, ... Resolving conflicts Open licenses Where to host work, and why

<https://training.github.com/kit/downloads/github-git-cheat-sheet.pdf>