software carpentry (http://software-carpentry.org)

Version Control with Git

(index.html)

Tracking Changes

✸ Learning Objectives

- Go through the modify-add-commit cycle for single and multiple files.
- Explain where information is stored at each stage of Git commit workflow.

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 (reference.html#commit) (or revision (reference.html#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 (http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html) 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?

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 +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 (reference.html#change-set) but not yet committed.

Staging area

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!

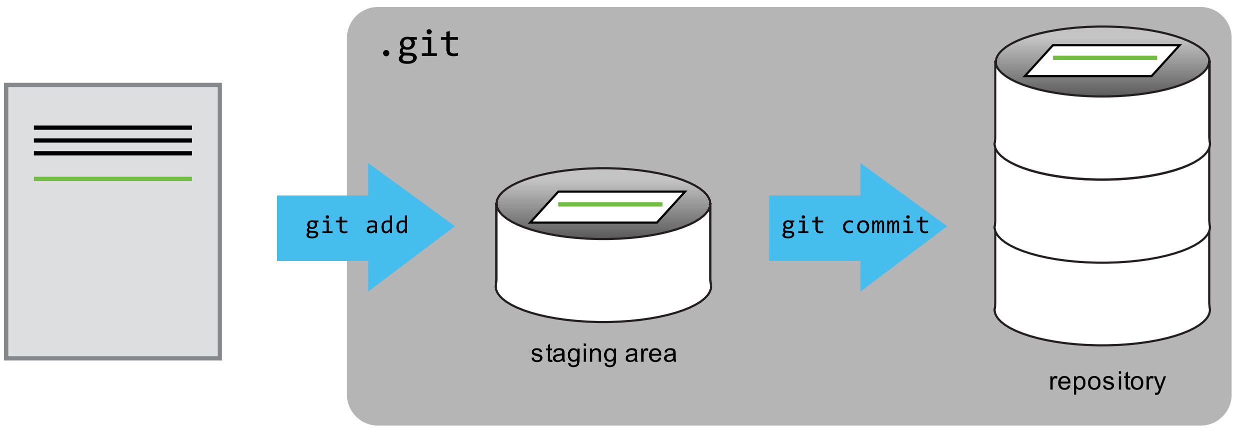

Figure: The Git Staging Area

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`):

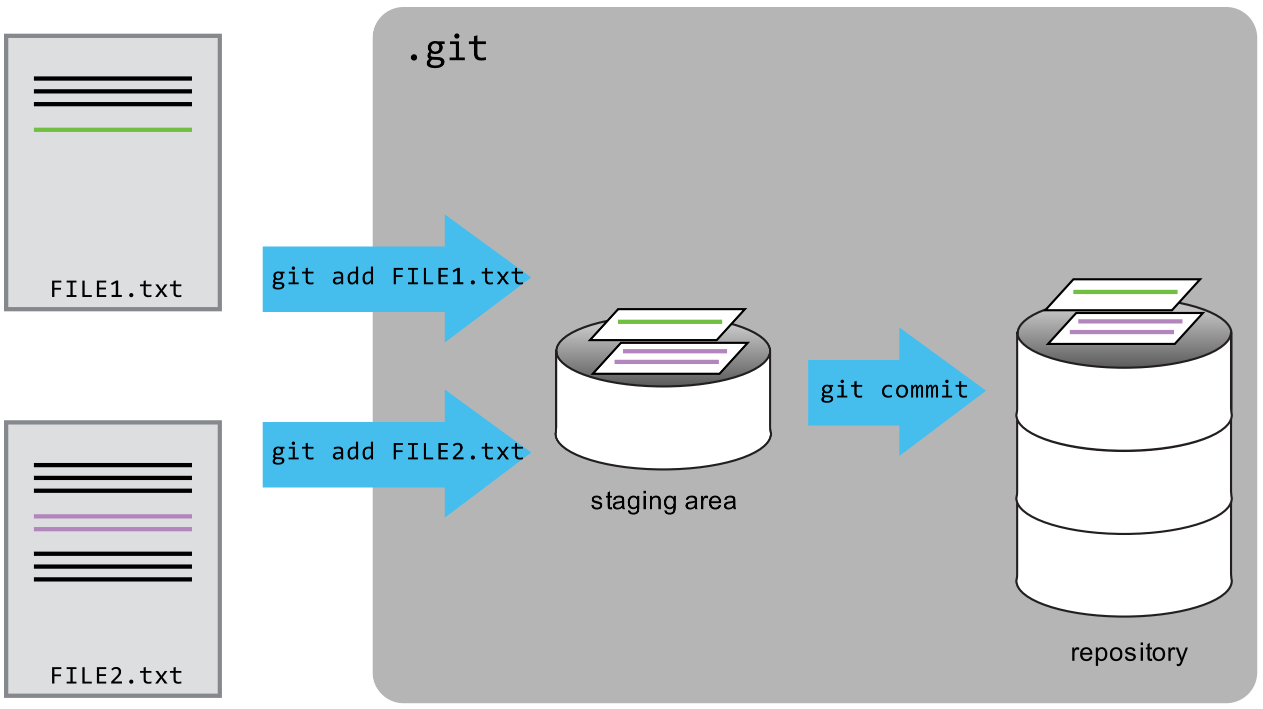

Figure: The Git Commit Workflow

Committing Changes to Git

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

```
$ git commit -m "my recent changes"
```

```
$ git init myfile.txt
$ git commit -m "my recent changes"
```

```
$ git add myfile.txt
$ git commit -m "my recent changes"
```

```
$ git commit -m myfile.txt "my recent changes"
```

`bio` Repository

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.