Version Control with Git

Git basics

Learning Objectives

- Configure `git` the first time is used on a computer.
- Create a local Git repository
- Understand the modify-add-commit cycle for a single file
- Configure Git to ignore specific files.

Getting started with Git

To learn about version control with Git, we are going to use the following scenario:

Wolfman and Dracula have been hired by Universal Missions to investigate if it is possible to send their next planetary lander to Mars. They want to be able to work on the plans at the same time, but they have run into problems doing this in the past. If they take turns, each one will spend a lot of time waiting for the other to finish, but if they work on their own copies and email changes back and forth things will be lost, overwritten, or duplicated. Therefore, they are going to use Git with Github to work on their plans at the same time, then merge their changes.

Git configuration

When we use Git on a new computer for the first time, we need to configure a few things. Below are a few examples of configurations we will set as we get started with Git:

- our name and email address,
- to colorize our output,
- what our preferred text editor is,
- and that we want to use these settings globally (i.e. for every project)

On a command line, Git commands are written as `git verb`, where `verb` is what we actually want to do. So here is how Dracula sets up his new laptop:

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

(Please use your own name and email address instead of Dracula's.)

He also has to set his favorite text editor, following this table:

Editor	Configuration command
nano	`$ git config --global core.editor "nano -w"`
Text Wrangler	`$ git config --global core.editor "edit -w"`
Sublime Text (Mac)	`$ git config --global core.editor "subl -n -w"`
Sublime Text (Win, 32-bit install)	`$ git config --global core.editor "'c:/program files (x86)/sublime text 3/sublime_text.exe' -w"`
Sublime Text (Win, 64-bit install)	`$ git config --global core.editor "'c:/program files/sublime text 3/sublime_text.exe' -w" >`
Notepad++ (Win)	`$ git config --global core.editor "'c:/program files (x86)/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlu`
Kate (Linux)	`$ git config --global core.editor "kate"`
Gedit (Linux)	`$ git config --global core.editor "gedit -s -w"`
emacs	`$ git config --global core.editor "emacs"`

Editor	Configuration command
vim	`$ git config --global core.editor "vim"`

The four commands we just ran above only need to be run once: the flag `--global` tells Git to use the settings for every project, in your user account, on this computer.

You can check your settings at any time:

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

You can change your configuration as many times as you want: just use the same commands to choose another editor or update your email address.

Proxy

In some networks you need to use a proxy (https://en.wikipedia.org/wiki/Proxy_server). If this is the case, you may also need to tell Git about the proxy:

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

To disable the proxy, use

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

Creating a local Git 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 (../reference.html#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)
```

Exercise

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` ? ***

Tracking changes with Git

Dracula creates a file called `mars.txt` that contains some notes about the Red Planet's suitability as a base. (We'll use `vim` to create and 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.)

```
$ vim 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 `vim` (or whatever other editor we configured as `core.editor`) so that we can write a longer message.

Good 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?

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.

```
$ vim mars.txt
```

Add the following line to the file, then save and quit:

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

```
$ cat mars.txt
```

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"
```

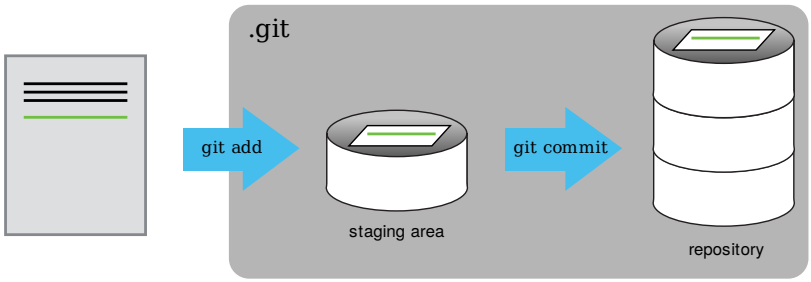
```
[master 34961b1] Add concerns about effects of Mars' moons on Wolfman
 1 file changed, 1 insertion(+)
```

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!



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:

```
$ vim 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
```

```
$ cat mars.txt
```

```
$ 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
```

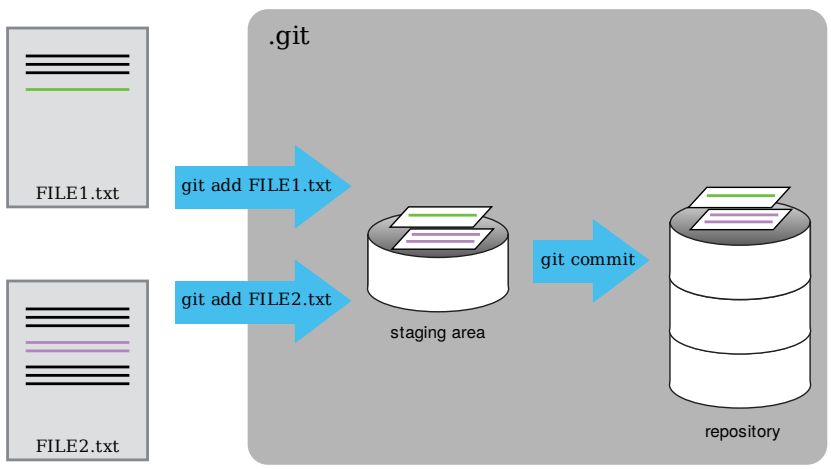
```
# On branch master
nothing to commit, working directory clean
```

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`):



The Git Commit Workflow

Exercises

1. Which of the following commit messages would be most appropriate for the last commit made to `mars.txt` ?
   a. “Changes”
   b. “Added line ‘But the Mummy will appreciate the lack of humidity’ to mars.txt”
   c. “Discuss effects of Mars’ climate on the Mummy”
2. Which command(s) below would save the changes of `myfile.txt` to my local Git repository?
   a. `$ git commit -m "my recent changes"`
   b. $ git init myfile.txt
      $ git commit -m “my recent changes”
   c. $ git add myfile.txt
      $ git commit -m “my recent changes”
   d. `$ git commit -m myfile.txt "my recent changes"`
3. 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.
4. 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 commiting 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.

Ignoring files during tracking

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` :

```
$ vim .gitignore
```

```
*.dat
results/
```

```
$ cat .gitignore
```

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
```

Exercises

1. Given a directory structure that looks like: ~~~ {.bash} results/data results/plots ~~~

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

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

3. 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?

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

```
*.data
!*.data
```

What will be the result?

5. You wrote a script that creates many intermediate log-files of the form log_01, log_02, log_03, etc. You want to keep them but you do not want to track them through `git`.

    a. Write **one** `.gitignore` entry that excludes files of the form `log_01`, `log_02`, etc.

    b. Test your "ignore pattern" by creating some dummy files of the form `log_01`, etc.

    c. You find that the file `log_01` is very important after all, add it to the tracked files without changing the `.gitignore` again.

    d. Discuss with your neighbor what other types of files could reside in your directory that you do not want to track and thus would exclude via `.gitignore`.

Comparing differences between files

If we want to see what we changed at different steps, we can use `git diff` again, but with the notation `HEAD~1`, `HEAD~2`, and so on, to refer to old commits:

```
$ git diff HEAD~1 mars.txt
```

```
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
```

```
$ git diff HEAD~2 mars.txt
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +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
```

In this way, we can build up a chain of commits. The most recent end of the chain is referred to as `HEAD`; we can refer to previous commits using the `~` notation, so `HEAD~1` (pronounced "head minus one") means "the previous commit", while `HEAD~123` goes back 123 commits from where we are now.

We can also refer to commits using those long strings of digits and letters that `git log` displays. These are unique IDs for the changes, and "unique" really does mean unique: every change to any set of files on any computer has a unique 40-character identifier. Our first commit was given the ID f22b25e3233b4645dabd0d81e651fe074bd8e73b, so let's try this:

```
$ git diff f22b25e3233b4645dabd0d81e651fe074bd8e73b mars.txt
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +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
```

That's the right answer, but typing out random 40-character strings is annoying, so Git lets us use just the first few characters:

```
$ git diff f22b25e mars.txt
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +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
```

Recovering Older Versions of a File

All right! So we can save changes to files and see what we've changed—now how can we restore older versions of things? Let's suppose we accidentally overwrite our file:

```
$ vim mars.txt
```

```
We will need to manufacture our own oxygen
```

```
$ cat mars.txt
```

`git status` now tells us that the file has been changed, but those changes haven't been staged:

```
$ 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")
```

We can put things back the way they were by using `git checkout` :

```
$ git checkout HEAD 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
```

As you might guess from its name, `git checkout` checks out (i.e., restores) an old version of a file. In this case, we're telling Git that we want to recover the version of the file recorded in `HEAD` , which is the last saved commit. If we want to go back even further, we can use a commit identifier instead:

```
$ git checkout f22b25e mars.txt
```

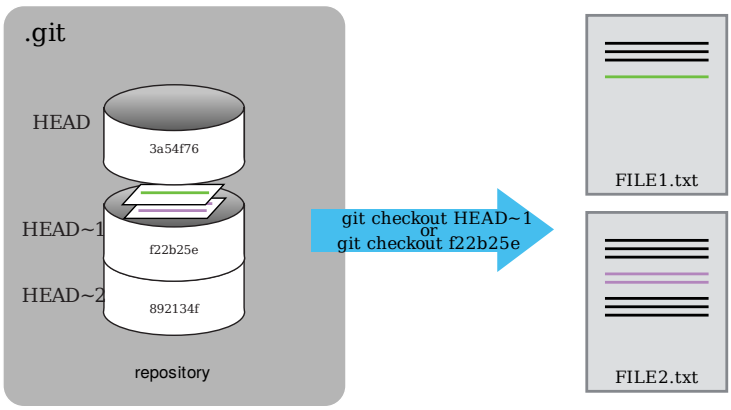
Don't lose your HEAD

Above we used

```
$ git checkout f22b25e mars.txt
```

to revert mars.txt to its state after the commit f22b25e. If you forget `mars.txt` in that command, git will tell you that "You are in 'detached HEAD' state." In this state, you shouldn't make any changes. You can fix this by reattaching your head using `git checkout master`

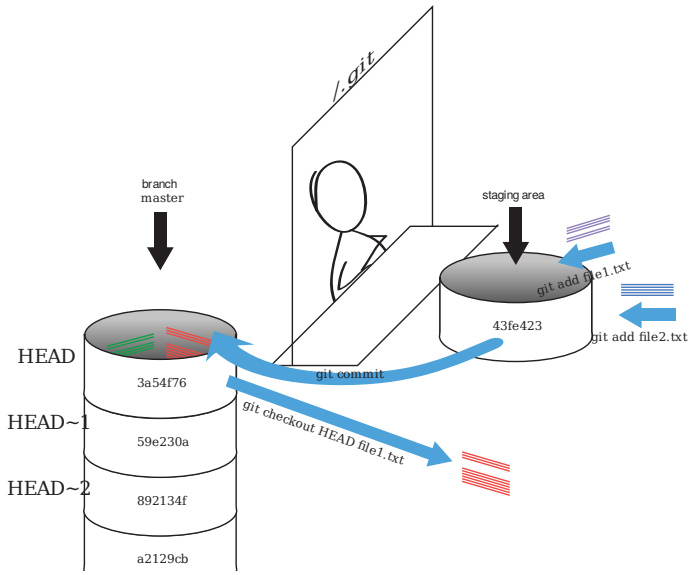
It's important to remember that we must use the commit number that identifies the state of the repository *before* the change we're trying to undo. A common mistake is to use the number of the commit in which we made the change we're trying to get rid of. In the example below, we want to retrieve the state from before the most recent commit (`HEAD~1`), which is commit `f22b25e` :



Git Checkout

So, to put it all together:

How Git works, in cartoon form



http://figshare.com/articles/How_Git_works_a_cartoon/1328266 (http://figshare.com/articles/How_Git_works_a_cartoon/1328266)

Simplifying the Common Case

If you read the output of `git status` carefully, you'll see that it includes this hint:

```
(use "git checkout -- <file>..." to discard changes in working directory)
```

As it says, `git checkout` without a version identifier restores files to the state saved in `HEAD`. The double dash `--` is needed to separate the names of the files being recovered from the command itself: without it, Git would try to use the name of the file as the commit identifier.

The fact that files can be reverted one by one tends to change the way people organize their work. If everything is in one large document, it's hard (but not impossible) to undo changes to the introduction without also undoing changes made later to the conclusion. If the introduction and conclusion are stored in separate files, on the other hand, moving backward and forward in time becomes much easier.

Exercises

1. Jennifer has made changes to the Python script that she has been working on for weeks, and the modifications she made this morning "broke" the script and it no longer runs. She has spent ~ 1hr trying to fix it, with no luck… Luckily, she has been keeping track of her project's versions using Git!

   Which of the options below will let her recover the last committed version of her Python script called `data_cruncher.py` ?

   a. `$ git checkout HEAD`

   b. `$ git checkout HEAD data_cruncher.py`

   c. `$ git checkout HEAD~1 data_cruncher.py`

   d. `$ git checkout <unique ID of last commit> data_cruncher.py`

   e. Both b & d

2. What is the output of `cat venus.txt` at the end of this set of commands?

```
$ cd planets
$ vim venus.txt #input the following text: Venus is beautiful and full of love
$ git add venus.txt
$ vim venus.txt #add the following text: Venus is too hot to be suitable as a base
$ git commit -m "comments on Venus as an unsuitable base"
$ git checkout HEAD venus.txt
$ cat venus.txt #this will print the contents of venus.txt to the screen
```

   a. `Venus is too hot to be suitable as a base`

   b. `Venus is beautiful and full of love`

   c. Venus is beautiful and full of love

      Venus is too hot to be suitable as a base

   d. Error because you have changed venus.txt without committing the changes