GIANT UNDO BUTTON

It's the first warm day of the year. All the snow has melted and you are out for a walk. With every step your good mood grows and the winter funk recedes. Everything is great and then you grind a freshly-thawed pile of dog poop into the sidewalk with your shoe. No problem. You remain calm and say with authority, "Command Z!" Nothing happens. There is no undo command for real life.

Git is something developers call "source control", but you can think of it as a giant undo button for your entire project. You should start every project in a Git repository. Why? Because Git will free your mind.

Have you ever saved a PSD with a name like client_name_final_FINAL_v2.psd? You save files with names like this to freeze a design at a good stopping point so you can try new ideas without ruining the work you've already done. As you start writing more CSS and HTML you will find your projects now consist of many files, not just one, and a simple naming convention won't work.

Imagine, you have just spent the last twenty minutes tweaking colors and nudging a logo into place and now you have an idea: what if the logo was on the left of the screen and really HUGE. You hesitate before writing the new code, because if you don't like the new logo it will be difficult to get it back to where you started from. But, if your files are in a Git repository you don't need to worry about losing all of your work. You can experiment with the new logo position and if you don't like it you can just rewind your work to a good point.

Git will let you try wild new design directions without worrying about losing your previous work.

Git Basics

The easiest way to use Git is to learn its command line interface. Okay, don't panic. I know I just casually mentioned the command line like it's No Big Deal, but seriously, this is going to be a piece of cake. You'll be working with only a few commands, you'll get in, get out and get on with your day.

In the case of Git, graphical interfaces do a bad job of hiding complexity — it's either too hidden or not hidden enough. So you won't know where to start and eventually you'll stumble into using advanced features without really knowing what they do. Soon enough you'll be so deep down the rabbit hole you'll think you've lost all of your files, (you probably haven't, it almost never happens when you are using Git). The only way to avoid the trap is to skip the GUI in the first place and stick to the basics. This does imply you will need to learn a little bit about the command line, but just enough to use Git. You don't need to master the command line.

Git has hundreds of commands, but you don't need to know nearly this many to be proficient. This chapter covers only the commands you need to get your job done. After the initial setup, you will find yourself only using two or three commands on a daily basis because some commands are only useful when you start a new project or when something goes wrong. Over time you will memorize most of these commands, but until then, this chapter is structured so you can come back and copy-paste commands when you need them.

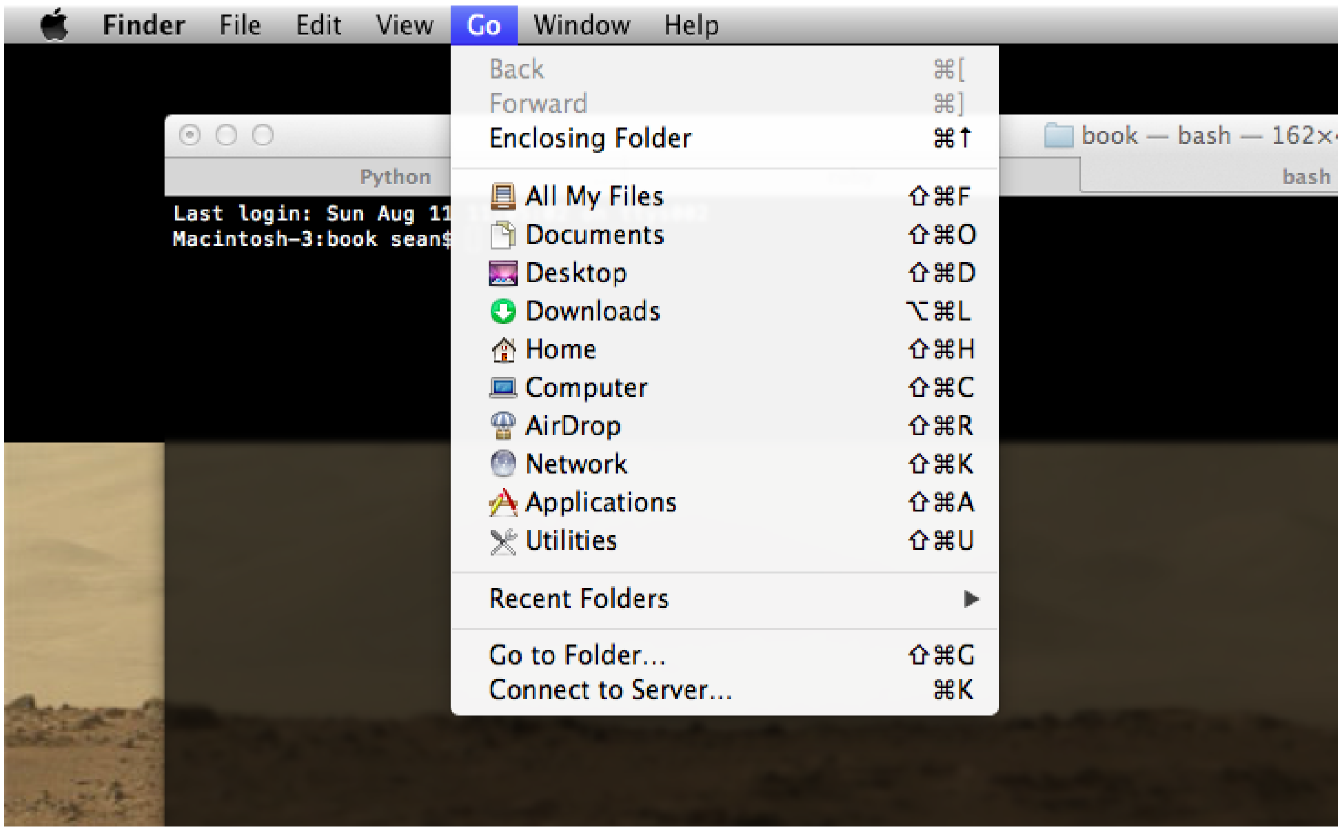
As you work your way through this chapter, and parts of the Sass chapter, remember: the command line is a critical tool in the workflow you are learning in this book. The whole point is to whittle down the tools front-end developers use to be exactly what you need as a designer, and no more. You need a toolchain and workflow that gets out of your way and lets you create results, in the browser, as fast as possible. When you get right down to it, the command line is simply more efficient than any GUI most of the time. But you can't get something for nothing, and all of the speed of the command line comes with the cost of an uncomfortable learning curve.

Installation

If you are using a Mac you can grab the latest version of Git [here.]() If you are on Windows [go here.]() Both links should start downloading the latest version of Git immediately. Choose the default options for each installer.

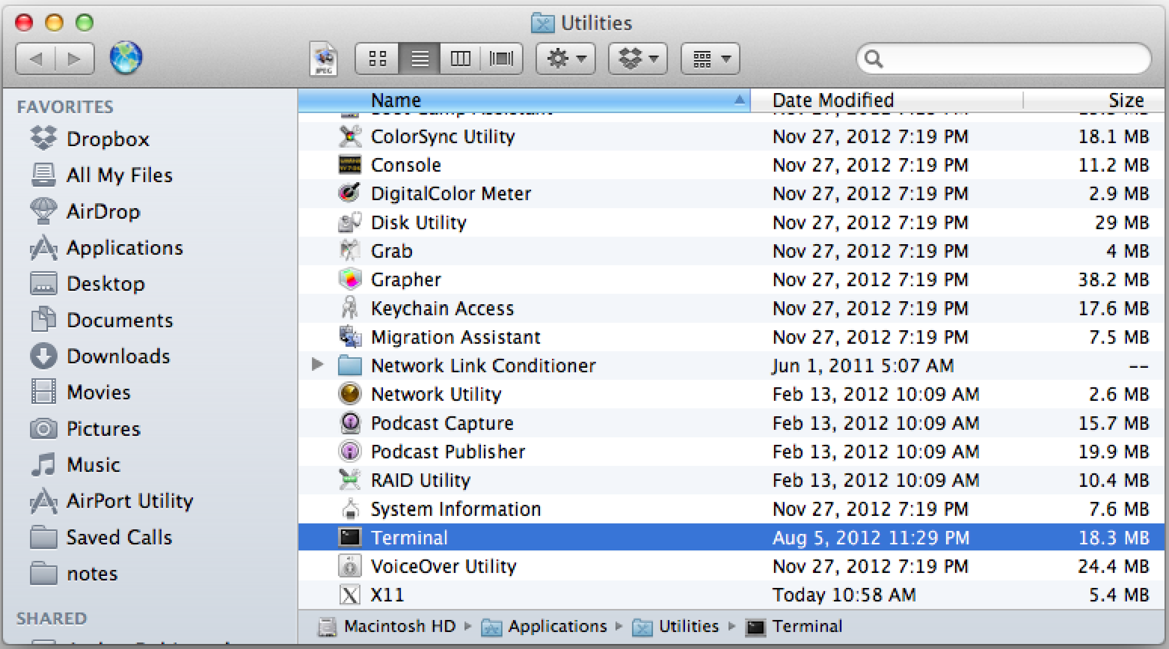
To test out your installation you need to open up the command line. Windows machines and Macs both have a command line built in, you just need to know how to find it.

On your Mac, open the Utilities folder and then run the Terminal application.



```
Last login: Sun Aug 11
Macintosh-3:book sean$
```

Get to the Utilities folder using the Go menu in the Finder.



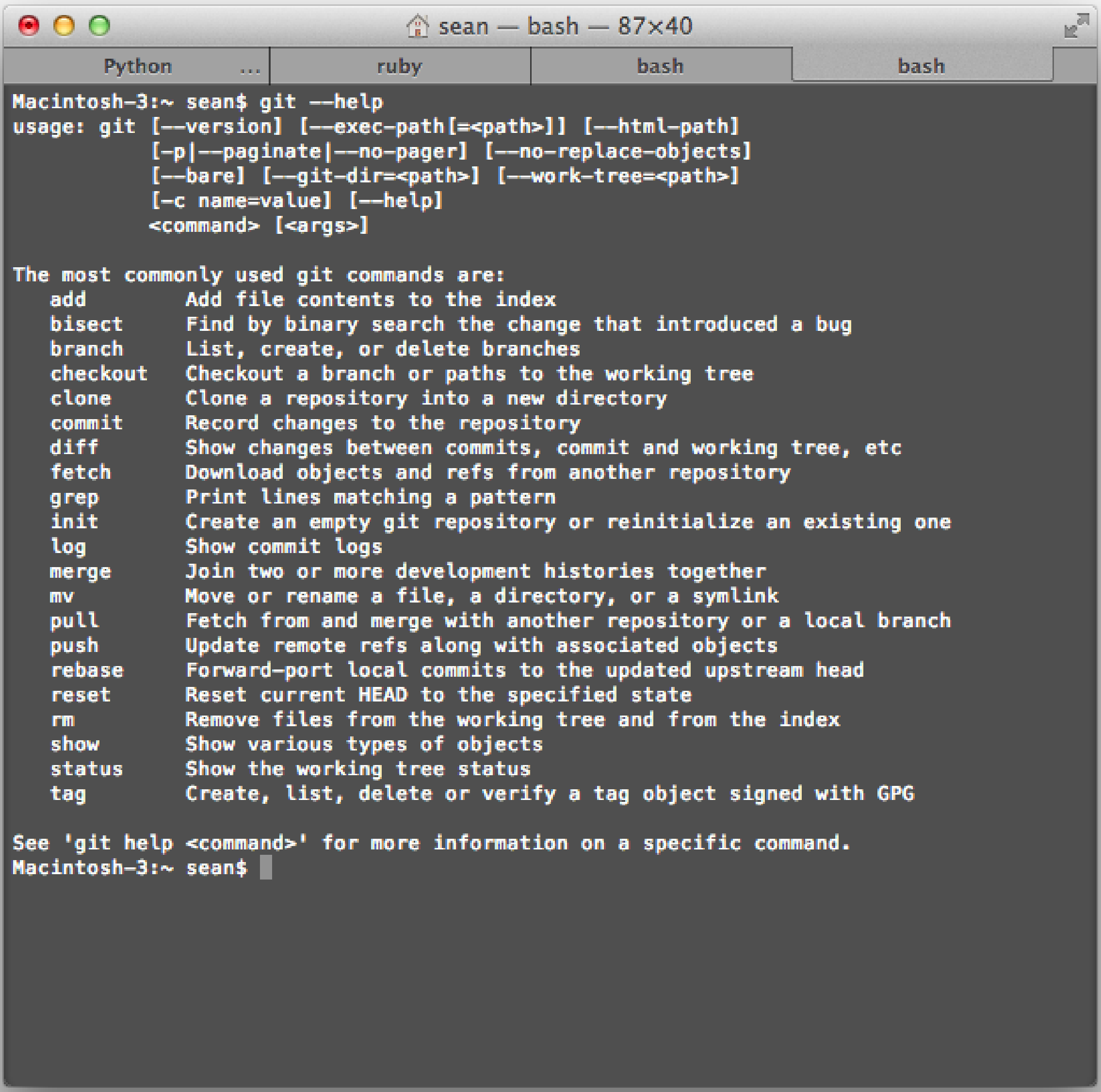
Name	Date Modified	Size
ColorSync Utility	Nov 27, 2012 7:19 PM	18.1 MB
Console	Nov 27, 2012 7:19 PM	11.2 MB
DigitalColor Meter	Nov 27, 2012 7:19 PM	2.9 MB
Disk Utility	Nov 27, 2012 7:19 PM	29 MB
Grab	Nov 27, 2012 7:19 PM	4 MB
Grapher	Nov 27, 2012 7:19 PM	38.2 MB
Keychain Access	Nov 27, 2012 7:19 PM	17.6 MB
Migration Assistant	Nov 27, 2012 7:19 PM	7.5 MB
Network Link Conditioner	Jun 1, 2011 5:07 AM	--
Network Utility	Feb 13, 2012 10:09 AM	2.6 MB
Podcast Capture	Feb 13, 2012 10:09 AM	15.7 MB
Podcast Publisher	Feb 13, 2012 10:09 AM	19.9 MB
RAID Utility	Feb 13, 2012 10:09 AM	10.4 MB
System Information	Nov 27, 2012 7:19 PM	7.6 MB
Terminal	Aug 5, 2012 11:29 PM	18.3 MB
VoiceOver Utility	Nov 27, 2012 7:19 PM	24.4 MB
X11	Today 10:58 AM	5.4 MB

Macintosh HD ▸ Applications ▸ Utilities ▸ Terminal

Open the Terminal application

In Windows you just need to type the letters "cmd" into the Run box in the Start Menu. So for Windows 7 and older you click Start and then in the text box at the bottom type "cmd" and hit enter. Windows 8 still has the run box, you just have to search for it.

With the command line open type git --help and hit enter. It should look something like this:



```
Macintosh-3:~ sean$ git --help
usage: git [--version] [--exec-path[=<path>]] [--html-path]
           [-p|--paginate|--no-pager] [--no-replace-objects]
           [--bare] [--git-dir=<path>] [--work-tree=<path>]
           [-c name=value] [--help]
           <command> [<args>]

The most commonly used git commands are:
   add        Add file contents to the index
   bisect     Find by binary search the change that introduced a bug
   branch     List, create, or delete branches
   checkout   Checkout a branch or paths to the working tree
   clone      Clone a repository into a new directory
   commit     Record changes to the repository
   diff       Show changes between commits, commit and working tree, etc
   fetch      Download objects and refs from another repository
   grep       Print lines matching a pattern
   init       Create an empty git repository or reinitialize an existing one
   log        Show commit logs
   merge      Join two or more development histories together
   mv         Move or rename a file, a directory, or a symlink
   pull       Fetch from and merge with another repository or a local branch
   push       Update remote refs along with associated objects
   rebase     Forward-port local commits to the updated upstream head
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index
   show       Show various types of objects
   status     Show the working tree status
   tag        Create, list, delete or verify a tag object signed with GPG

See 'git help <command>' for more information on a specific command.
Macintosh-3:~ sean$ 
```

Git's help command. If you see this, everything is installed correctly.

Now that everything is installed it doesn't matter if you are on a Mac or on Windows, the commands are pretty universal. And keep the command line open, you'll be using it again shortly.

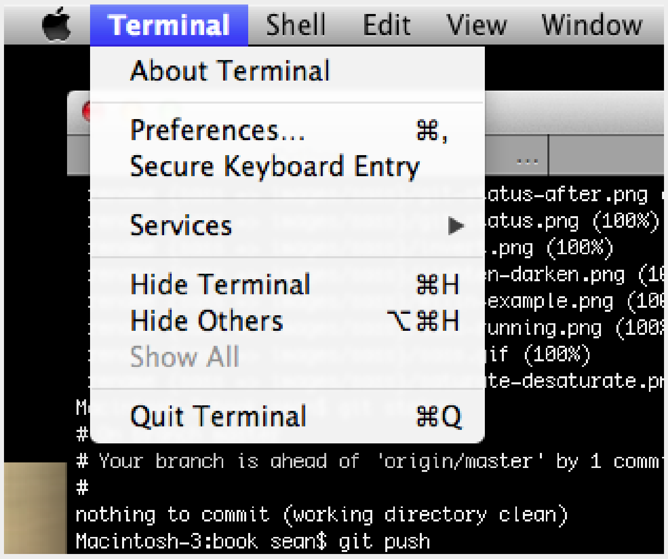
Change the command line colors

The screenshots in this chapter are all done on a Mac. If you are on a Mac, and want your command line to look exactly like the screenshots in the chapter, you need to change the color scheme. Windows users are out of luck. The command line there is not as flexible, but the textual output from the Git commands will be the same.

First you need to tell Git to output colors. Type this in an open command line window:

```
git config --global color.ui true
```

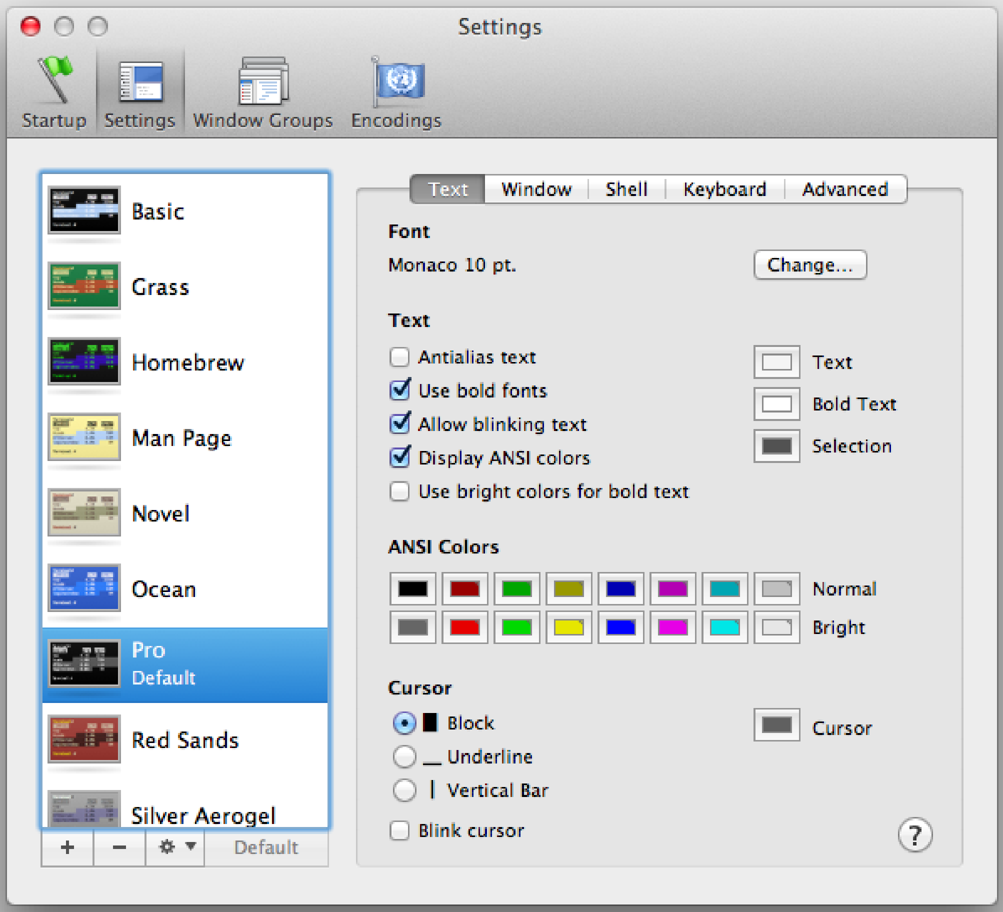
Now you need to configure your terminal window to use the same style as the examples. On your Mac, go to your terminal window, open up the Terminal menu and go to Preferences.



```
atus-after.png
atus.png (100%)
.png (100%)
n-darken.png (1
example.png (10
unning.png (100
if (100%)
te-desaturate.p
M
#
# Your branch is ahead of 'origin/master' by 1 comm
#
nothing to commit (working directory clean)
Macintosh-3:book sean$ git push
```

Go to Preferences in the Terminal menu

In the Preferences window, click on Settings and in the left box, click on the Pro option. With the Pro option highlighted, click on the Default button at the bottom of all of the options. Now if you open a new Terminal window it should use the new Pro color scheme.



Font
Monaco 10 pt.

Text
- [ ] Antialias text
- [✓] Use bold fonts
- [✓] Allow blinking text
- [✓] Display ANSI colors
- [ ] Use bright colors for bold text

Text / Bold Text / Selection

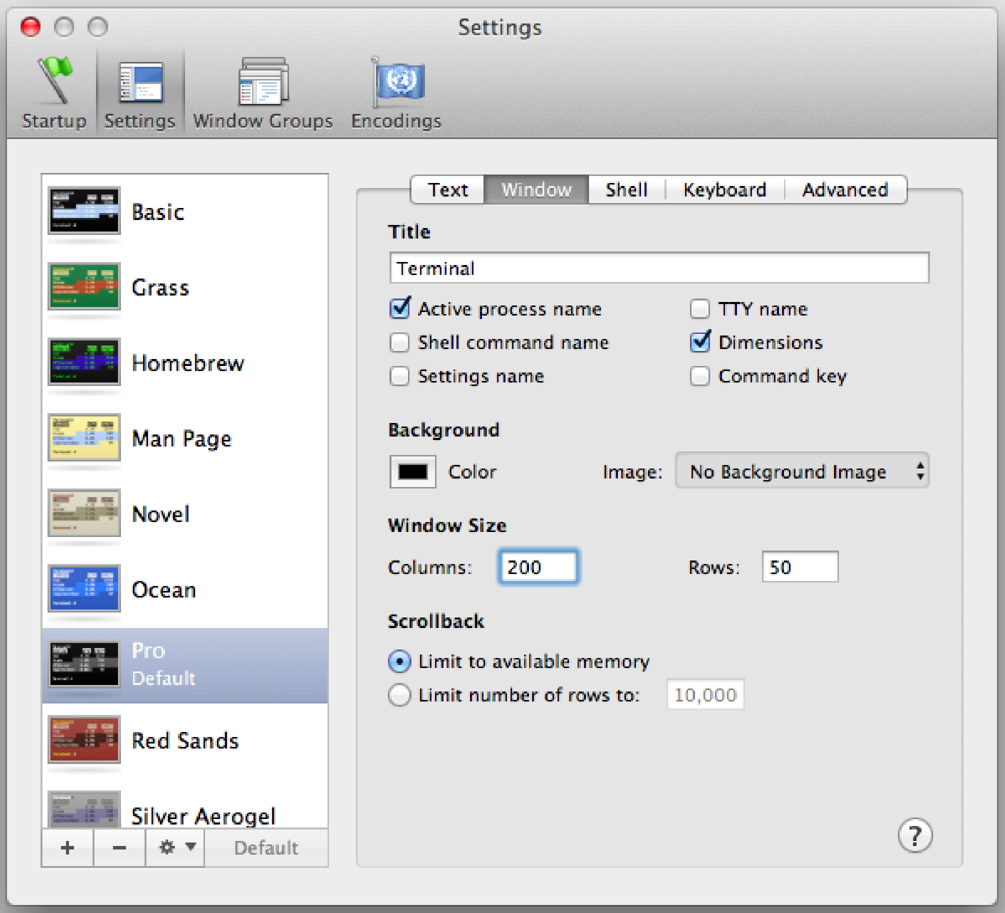
ANSI Colors

Cursor
- [✓] Block
- [ ] Underline
- [ ] Vertical Bar
- [ ] Blink cursor

Cursor

Choose the Pro option

You might also consider making your terminal window open larger by default. In the same Preferences, with Pro highlighted, choose the "Window" option. In the "Window Size" section you can adjust the default width, (columns), and the height, (row), of the terminal window, 150-200 columns and 40 rows works well.



Title

Terminal

[✓ Active process name] TTY name
Shell command name [✓ Dimensions]
Settings name Command key

Background

Color Image: No Background Image

Window Size

Columns: 200 Rows: 50

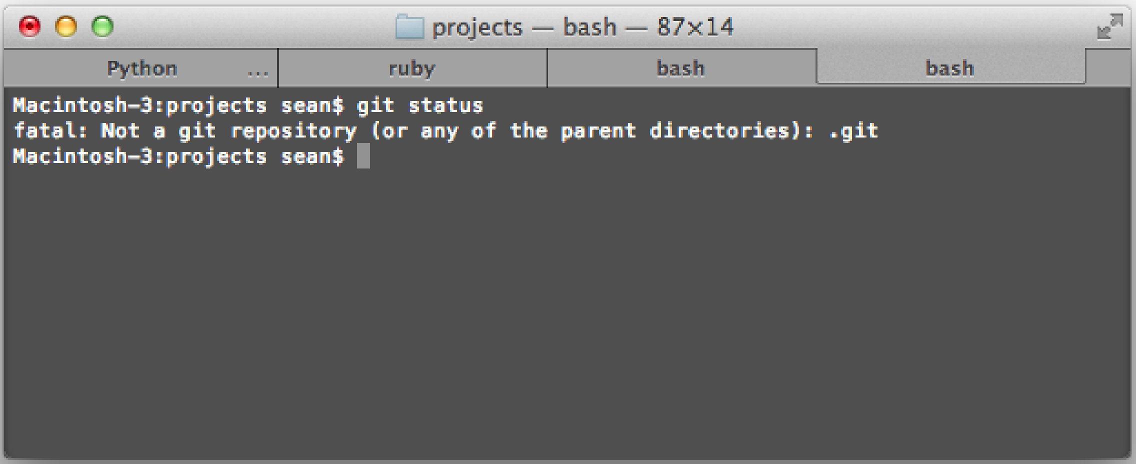
Scrollback

[✓ Limit to available memory]
Limit number of rows to: 10,000

Set the columns and rows to whatever values you prefer.

Create a repository

Your code will live in a Git repository. A repository is just a regular folder that contains your project's code plus some special files Git uses to keep things organized. You'll never look at the special files, (they are hidden by default), but Git needs them. In fact, Git doesn't know what to do unless they exist. Try running a Git command right now, before you have created a repository. Type git status into the command line.



```
Macintosh-3:projects sean$ git status
fatal: Not a git repository (or any of the parent directories): .git
Macintosh-3:projects sean$
```

If you try to run Git commands in a folder that isn't a repository Git complains.

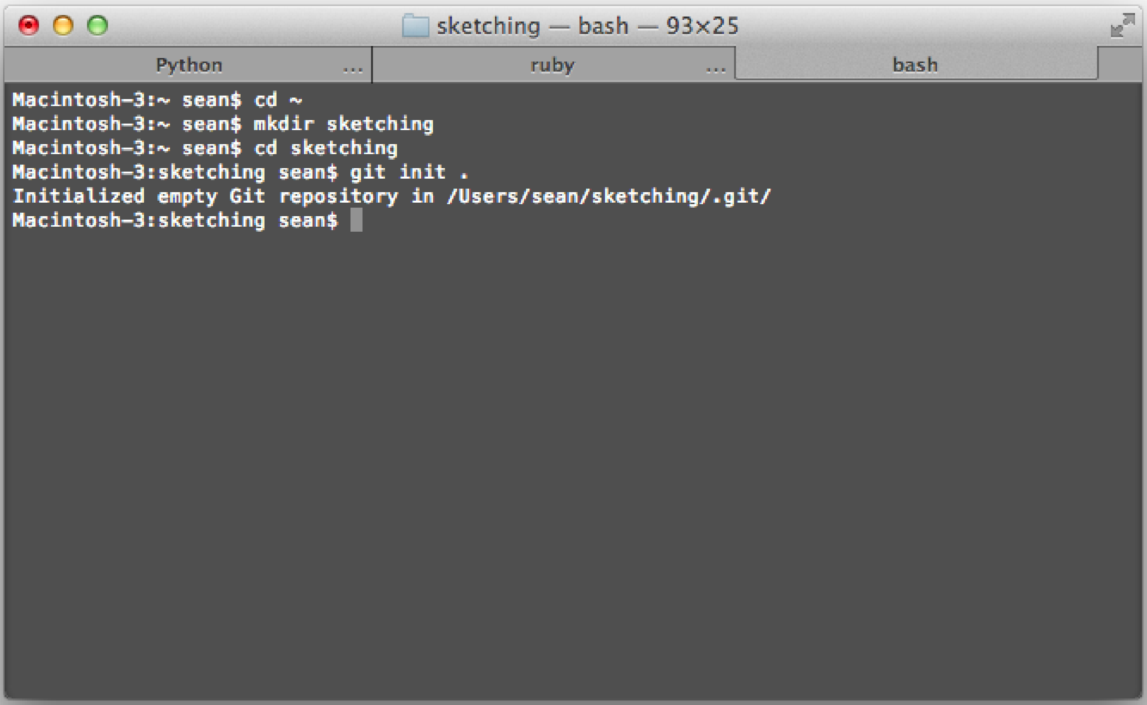
It didn't work because it's just a regular folder on your computer, not a Git repository. Turning a folder into a Git repository is easy, but first you need a folder. In your command line window type these commands. Pay attention to spaces. *Spaces matter!*

```
cd ~
    mkdir sketching
    cd sketching
    git init .
```

(In Windows type cd / instead of cd ~).

What just happened? When you type cd ~ you moved the prompt into the home folder — cd stands for "change directory" and ~ is shorthand for the full path to your home directory. mkdir stands for "make directory", (directory is just another word for folder), and so mkdir sketching just created a folder called sketching. You then cd'd into the new folder and told Git to make it a repository using the init command.

You will need to do this exactly once for every project. Any time you create a new project, repeat these steps and just use a different name instead of 'sketching'. Here is what it looks like in the command line:



```
Macintosh-3:~ sean$ cd ~
Macintosh-3:~ sean$ mkdir sketching
Macintosh-3:~ sean$ cd sketching
Macintosh-3:sketching sean$ git init .
Initialized empty Git repository in /Users/sean/sketching/.git/
Macintosh-3:sketching sean$
```

Congratulations, you've just created your first Git repository! It's in the folder called "sketching" in your home directory. For future reference, anytime you open up a new terminal window and want to get back into your repository, you need to cd into the folder you created, like so:

```
cd ~/sketching
```

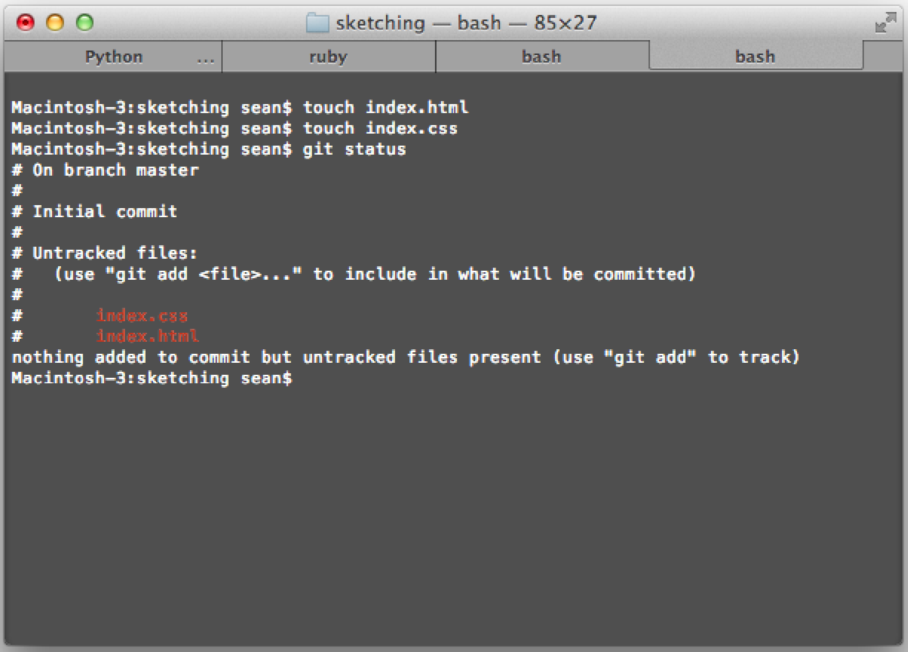
Add files to your repository

Right now you have an empty folder so you need to add some files:

```
touch index.html
   touch index.css
```

The touch command creates an empty file with the specified name. You could also create these files by opening up your text editor and saving new files to the "sketching" folder you created. The touch command is just a convenient way to create a new file when you are already in the command line.

One of the Git commands you will use often is status which tells you about the state of the files in your git repository. What does Git think about the new files you just created? Type git status into the command line to find out.

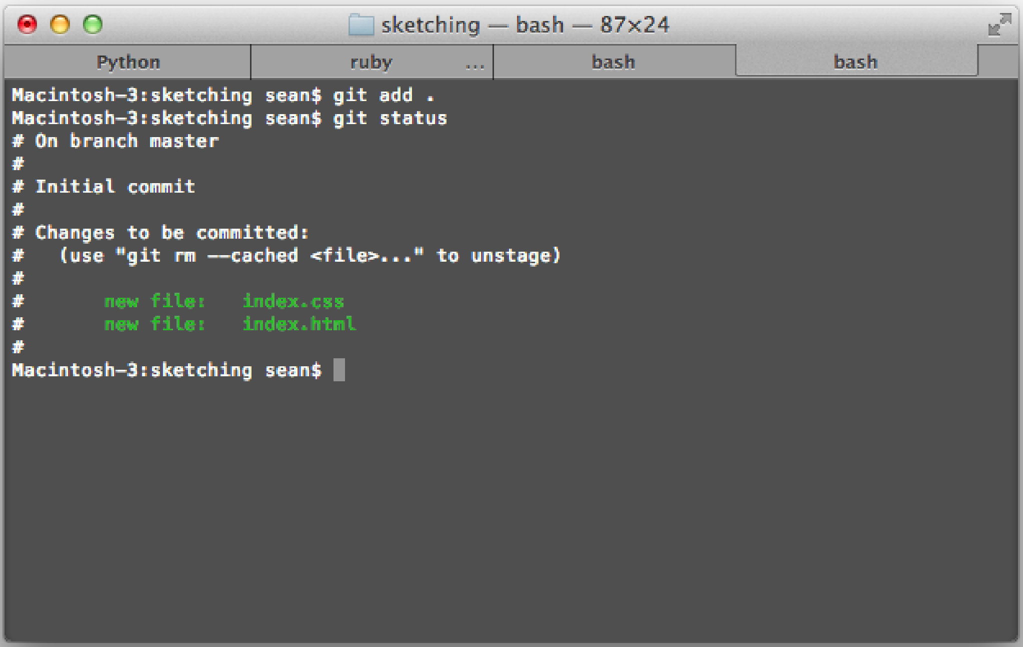


```
Macintosh-3:sketching sean$ touch index.html
Macintosh-3:sketching sean$ touch index.css
Macintosh-3:sketching sean$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       index.css
#       index.html
nothing added to commit but untracked files present (use "git add" to track)
Macintosh-3:sketching sean$
```

The names of the two files you added are in the "Untracked Files" section, (and may also be red if you've configured your terminal to use the same color settings used in the screenshots).

It turns out Git does not keep track of files automatically, you have to explicitly add them to the repository. This concept is a little confusing the first time you run into it. You've just created two files in the repository folder, but they aren't in the repository yet — there can be files in the folder that *aren't in the repository* and Git will ignore any changes you make to them. The "repository" consists of the files in the folder you decide should be tracked, therefore you must explicitly tell Git what files to add to the repository. To add files to the repository use add. To add everything in the current folder, (a common thing to do), just put a period after the add command, like so:

```
git add .
```



```
Macintosh-3:sketching sean$ git add .
Macintosh-3:sketching sean$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   index.css
#       new file:   index.html
#
Macintosh-3:sketching sean$
```

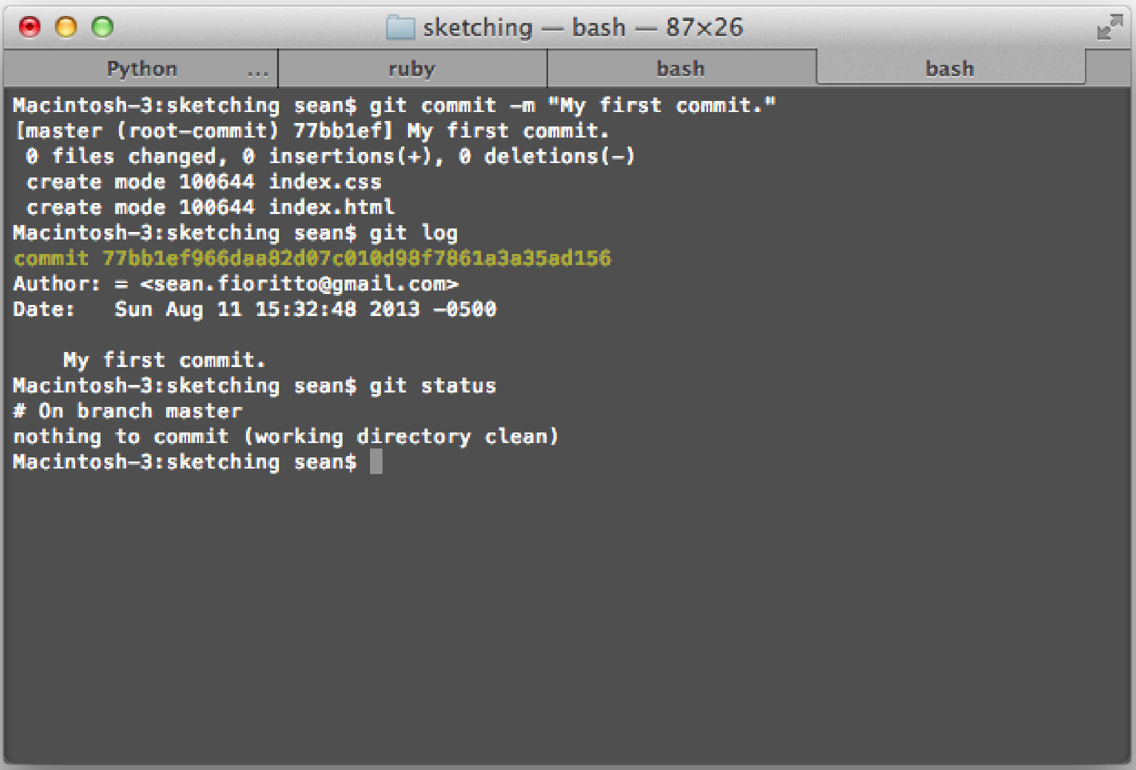
The result of adding files.

The status command now shows the files as green under a section called "Changes to be committed". Green files are tracked red files are not. What does it mean to be tracked? When you want to save changes in Git you have to do two things: you have to tell Git what you want to save and then you actually make the save. You use add to specify the stuff you want to save and commit to save the changes. Why split saving into two steps? Let's say you were in the middle of making some changes and you liked what you did in one file but not the other, by splitting the save function into two steps Git lets you preserve the changes from only the good file.

Keep things simple: add everything in the folder before you commit, (git add .), every time.

Even though the files are green they aren't saved yet because we haven't done the second step which is to commit the changes. With the next set of commands you will commit your changes and call the commit, "My first commit".

```
git commit -m "My first commit."
   git log
   git status
```



```
Macintosh-3:sketching sean$ git commit -m "My first commit."
[master (root-commit) 77bb1ef] My first commit.
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 index.css
 create mode 100644 index.html
Macintosh-3:sketching sean$ git log
commit 77bb1ef966daa82d07c010d98f7861a3a35ad156
Author: = <sean.fioritto@gmail.com>
Date:   Sun Aug 11 15:32:48 2013 -0500

    My first commit.
Macintosh-3:sketching sean$ git status
# On branch master
nothing to commit (working directory clean)
Macintosh-3:sketching sean$
```

Completing the save.

Terminal stuck after entering git log?

After you type git log, if the output is bigger than your terminal window, the terminal may dump you intoa special mode for scrolling through long output. You can navigate the output with the up or down arrows or the space bar to skip through pages.

When you are done reading the output, just hit `q` to exit the special mode and get back to the regular command line.

The commit command has extra options you can use, and that's what the "-m" bit is, it's just an option. In this case it specifies a message that describes what you are saving. The message is in quotes next to the "-m" flag, (that's what these kind of options are called). Every save in Git has a message that is saved in the log. This is incredibly useful. The log serves as a description for each change you've made. If you ever need to go back in time to a change you made you just skim the log and use the descriptions to find what you are looking for. So be kind to your future self and put a little thought into these messages.

If you want to see the log just use the command git log. This will display a list of all the changes you have made and their descriptions.

After committing the changes status shows there is nothing to commit. There are no outstanding changes in the repository.

Changing files

You have now created a new repository and added files to it. Time to make some changes. Let's add a doctype to the html file and set the base font color in the CSS file.

Add this to index.html:

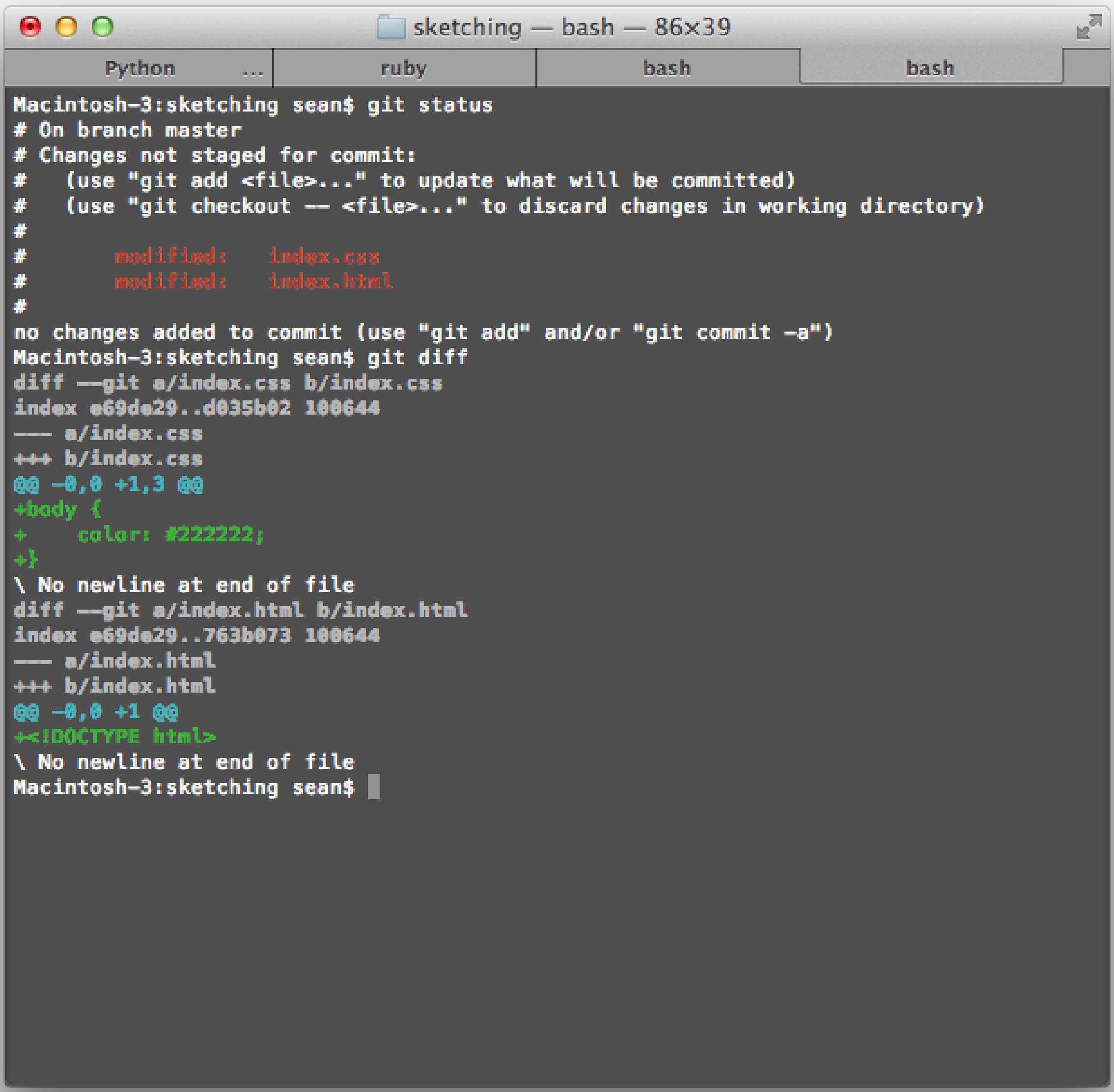
```
<!DOCTYPE html>
```

Add this to index.css:

```
body {
      color: #222222;
   }
```

Once you've done that, save those changes in your text editor and go back to the command line.

```
git status
   git diff
```



```
Macintosh-3:sketching sean$ 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:   index.css
#       modified:   index.html
#
no changes added to commit (use "git add" and/or "git commit -a")
Macintosh-3:sketching sean$ git diff
diff --git a/index.css b/index.css
index e69de29..d035b02 100644
--- a/index.css
+++ b/index.css
@@ -0,0 +1,3 @@
+body {
+    color: #222222;
+}
\ No newline at end of file
diff --git a/index.html b/index.html
index e69de29..763b073 100644
--- a/index.html
+++ b/index.html
@@ -0,0 +1 @@
+<!DOCTYPE html>
\ No newline at end of file
Macintosh-3:sketching sean$
```

Git can show you what changes you've made.

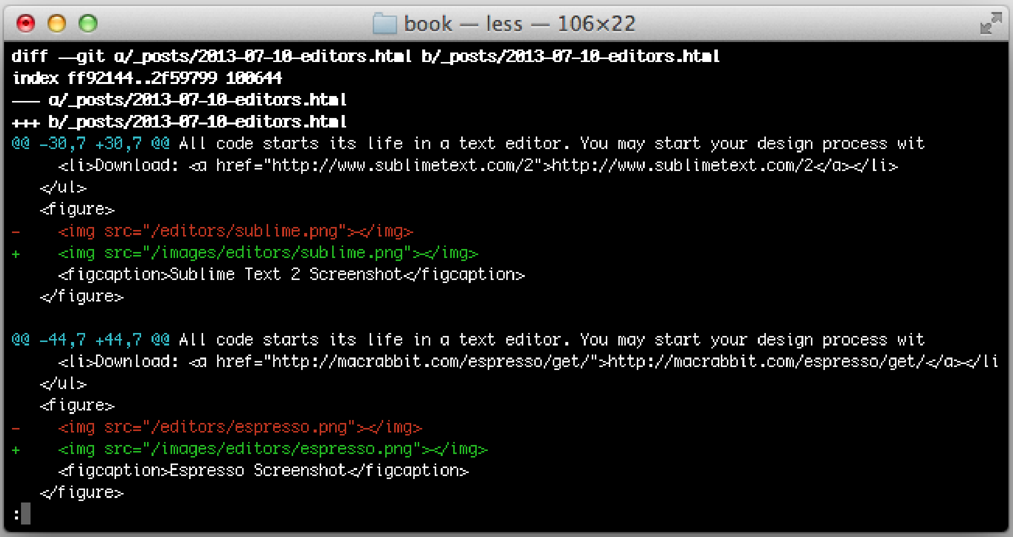
The status command tells you two files have been changed. The diff command shows you the changes you have made in greater detail. In this case you just made two small changes, so diff is not that useful. But most of the time you will be making quite a few changes before you decide to commit so diff lets you review the changes before you commit them.

Before you move on to the next section, be sure to save the changes you just made to the files.

```
git add .
   git commit -m "Added a doctype and chose a color."
```

Did your terminal window get stuck at git diff?

If the output of a command is bigger than the terminal screen, the terminal window goes into a special mode.



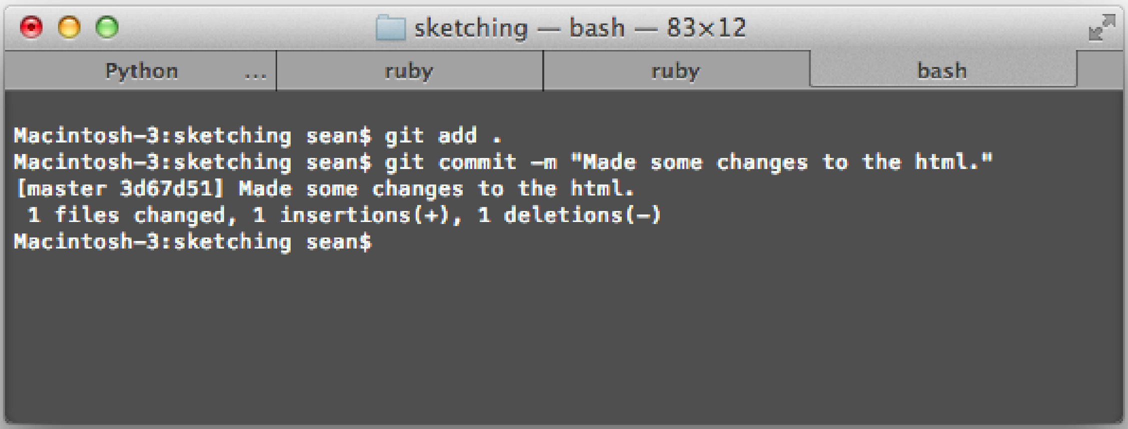
```
diff --git a/_posts/2013-07-10-editors.html b/_posts/2013-07-10-editors.html
index ff92144..2f59799 100644
--- a/_posts/2013-07-10-editors.html
+++ b/_posts/2013-07-10-editors.html
@@ -30,7 +30,7 @@ All code starts its life in a text editor. You may start your design process wit
     <li>Download: <a href="http://www.sublimetext.com/2">http://www.sublimetext.com/2</a></li>
   </ul>
   <figure>
-    <img src="/editors/sublime.png"></img>
+    <img src="/images/editors/sublime.png"></img>
     <figcaption>Sublime Text 2 Screenshot</figcaption>
   </figure>

@@ -44,7 +44,7 @@ All code starts its life in a text editor. You may start your design process wit
     <li>Download: <a href="http://macrabbit.com/espresso/get/">http://macrabbit.com/espresso/get/</a></li
   </ul>
   <figure>
-    <img src="/editors/espresso.png"></img>
+    <img src="/images/editors/espresso.png"></img>
     <figcaption>Espresso Screenshot</figcaption>
   </figure>
:
```

Look closely at the bottom, see the colon with the cursor next to it?

If your output looks cut-off and you see a colon followed by a cursor at the bottom of the window, then you are in a special "buffered" mode. From here you can use the arrow keys to scroll up and down, or you can hit the space key to go to the end. To get out of the mode and back to the command line, type the letter 'q'.

Information overload? All together now...

If you are feeling a little overwhelmed don't hit the panic button just yet. What you have read to this point is mostly for reference and is much more complicated than what you will be doing day-to-day. Here is what your typical interaction with Git will look like after you have made a few changes to some files:



```
Macintosh-3:sketching sean$ git add .
Macintosh-3:sketching sean$ git commit -m "Made some changes to the html."
[master 3d67d51] Made some changes to the html.
 1 files changed, 1 insertions(+), 1 deletions(-)
Macintosh-3:sketching sean$
```

A typical Git session.

90% of the time all you will do is type in these two commands and, presto-chango, you get most of the benefit of using Git. If you want to get fancy, before you add changes do a git diff to review what you have done and then before you commit do a git status to make sure you have added all of the files you want to commit.

Here are the commands in plain text so you can copy-paste them into the command line:

```
git status
```

Use this command before you do anything to make sure you know exactly what state your repository is in, e.g. are there other files you changed that you forgot about?

```
git diff
```

Optionally, after running git status you can run git diff to get a more detailed look at the changes you are about to commit.

```
git add .
```

Add all of the changes you've made plus any new files so they are ready to be committed. If you run git status after git add . then you should see every file as green.

```
git commit -m "put a custom message between these quotes"
```

This saves your changes permanently in the git repository. Now you never have to worry about losing your work. You are free to experiment.

Swinging from the branches

In Photoshop you can add layers, make changes to those layers and then remove them without affecting the other layers. Git has a similar mechanism called a branch. A branch in Git is like a Photoshop layer except it's for your HTML and CSS. Want to try a completely new color scheme? Create a branch, start coding and if you don't like it, flip back to where you started and delete the branch. A branch lets you try something new and then revert back to where you started whenever you want, or keep your changes.

Creating a branch is pretty easy. Hop back into your command line and type this:

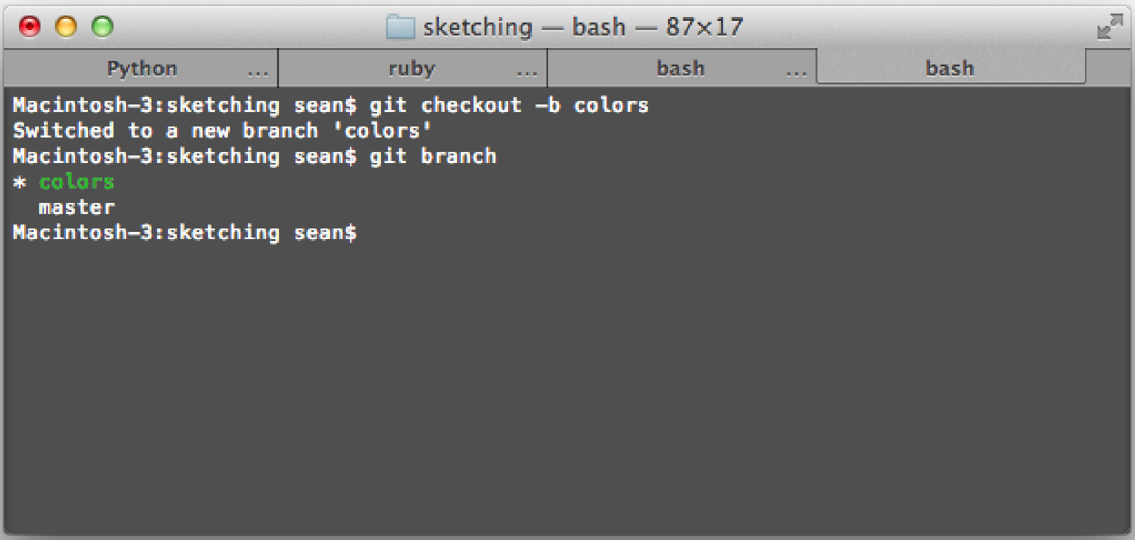
```
git checkout -b colors
```

This is actually two commands in one. The base command, git checkout, switches the view over to the new branch. The -b colors option actually creates the new branch, called colors, for you to switch to. In Photoshop, when you create a new layer you need to switch to that layer in order for your new changes to happen in that layer. Branches are the same. You can switch back and forth between branches in Git using the git checkout command without the -b option, e.g. to get back to the master branch from where you are now you type git checkout master.

There is a big difference between layers and branches. Layers in Photoshop are visible by default, in Git, you don't see the changes in a branch unless the branch is active. So if you switch back and forth between branches, you won't see changes from one branch in another branch.

The next step is to make some changes on the new "colors" branch and then merge them back into the master branch. First take a look at all the branches in your repository.

```
git branch
```



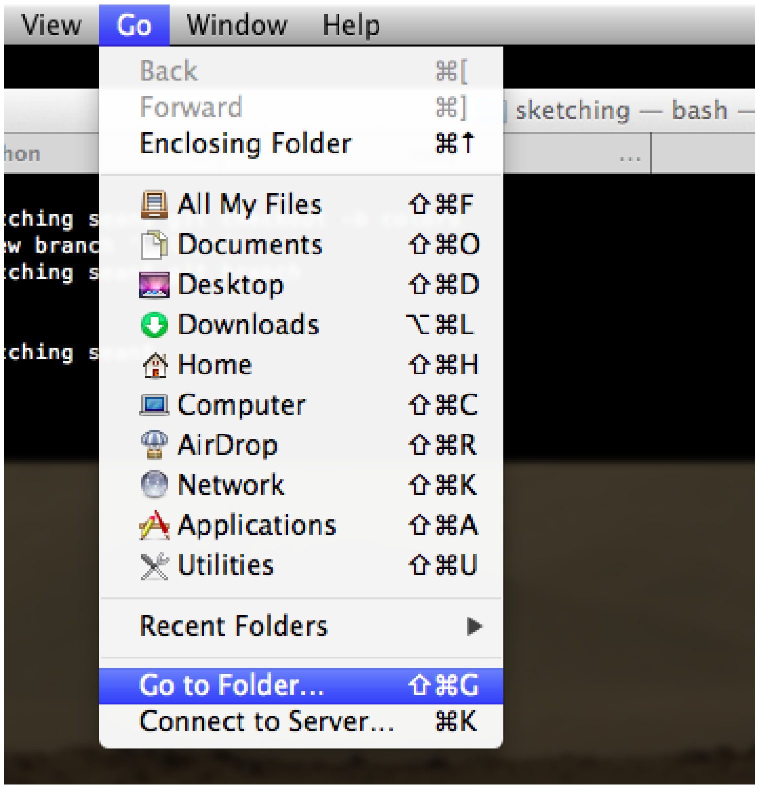
```
Macintosh-3:sketching sean$ git checkout -b colors
Switched to a new branch 'colors'
Macintosh-3:sketching sean$ git branch
* colors
  master
Macintosh-3:sketching sean$
```

A list of the branches in your repository.

The command git branch lists all of the branches in your repository. The active, visible branch is green and has an asterisk next to it. If you make changes in this branch they will only be in this branch, they will not show up anywhere else.

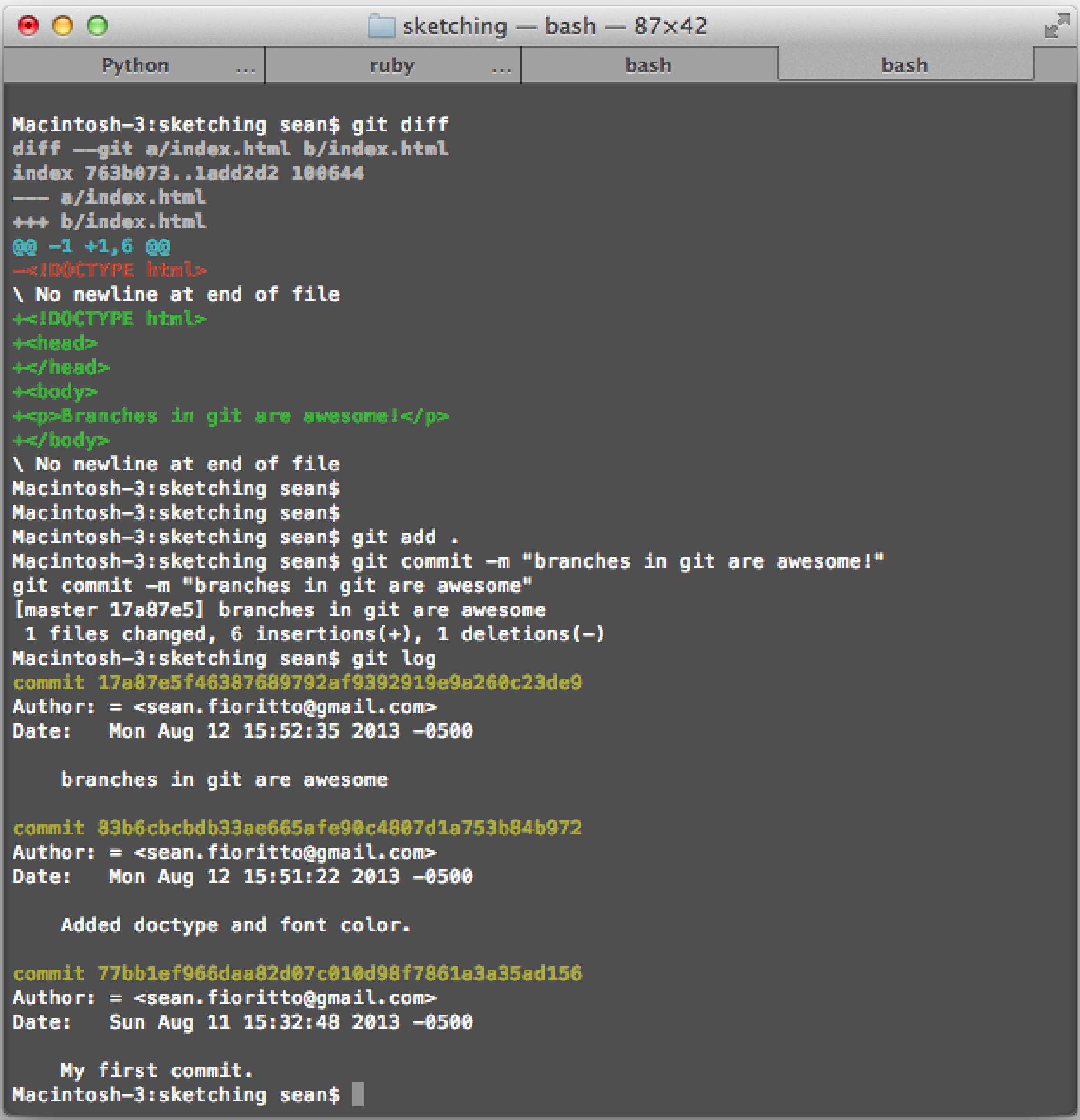
There is another branch in this list called "master". Every Git repository has a master branch by default. The master branch represents the canonical, stable branch of code. The other branches are experiments. When you are done with an experiment you merge those changes back into the master branch.

Go ahead and edit one of the files in your repository. If you are in OSX and don't know where your home directory is, (which is where your repository is located if you followed the steps at the beginning of this chapter), go to the command line and type pwd. This will list the full path to the repository. Copy/paste that path into the Go to Folder window (hit Command-Shift-G or find it in the Go menu). Open one of those files and make some changes. (If you are in Windows and followed the instructions in this chapter, your repository is in the C: drive at the top level).



Go to Folder

Save your changes and then commit them. Your commit will be on the new branch. Take a look at the log and you can see your new commit.



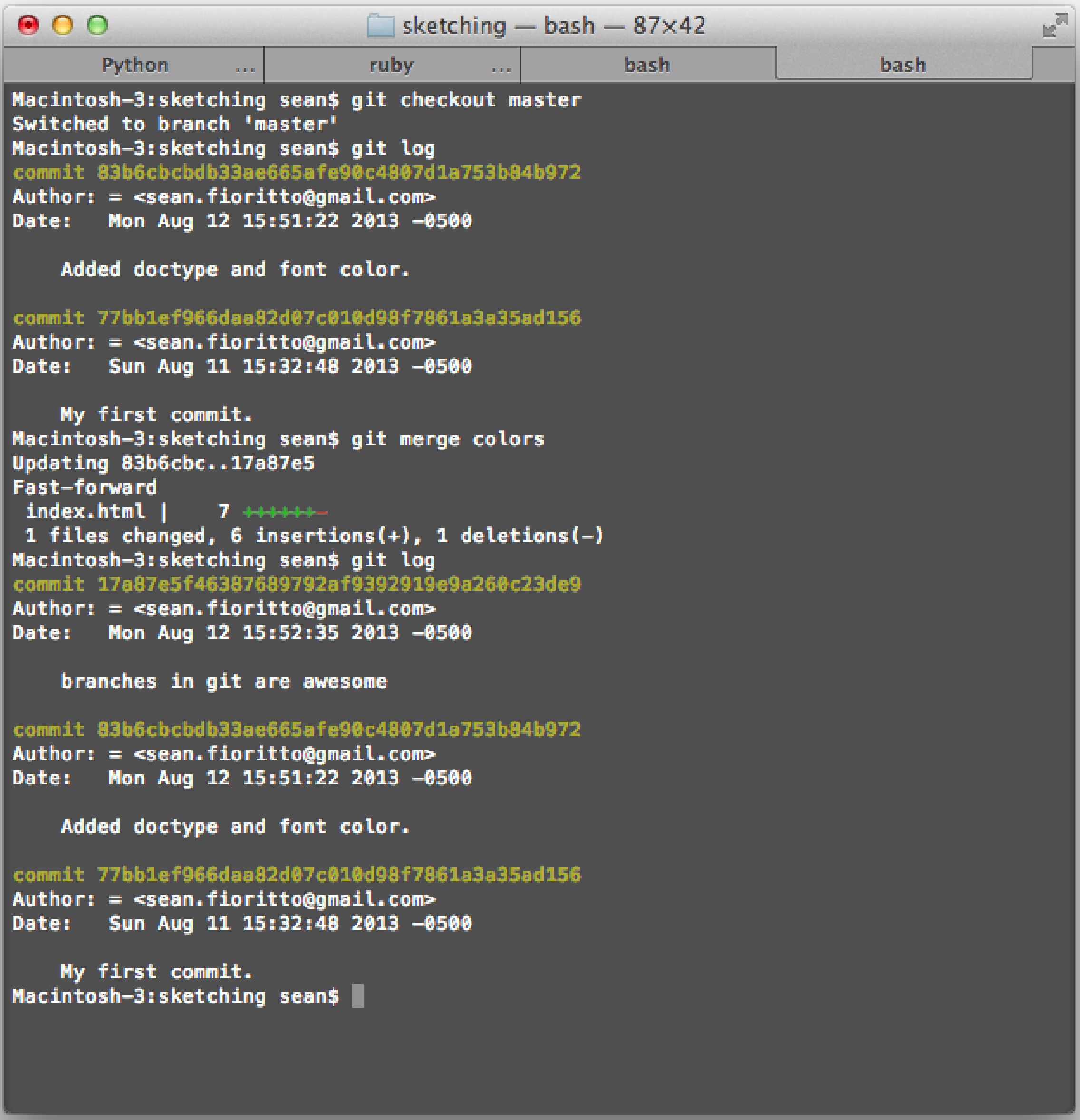
```
Macintosh-3:sketching sean$ git diff
diff --git a/index.html b/index.html
index 763b073..1add2d2 100644
--- a/index.html
+++ b/index.html
@@ -1 +1,6 @@
-<!DOCTYPE html>
\ No newline at end of file
+<!DOCTYPE html>
+<head>
+</head>
+<body>
+<p>Branches in git are awesome!</p>
+</body>
\ No newline at end of file
Macintosh-3:sketching sean$
Macintosh-3:sketching sean$
Macintosh-3:sketching sean$ git add .
Macintosh-3:sketching sean$ git commit -m "branches in git are awesome!"
git commit -m "branches in git are awesome"
[master 17a87e5] branches in git are awesome
 1 files changed, 6 insertions(+), 1 deletions(-)
Macintosh-3:sketching sean$ git log
commit 17a87e5f46387689792af9392919e9a260c23de9
Author: = <sean.fioritto@gmail.com>
Date:   Mon Aug 12 15:52:35 2013 -0500

    branches in git are awesome

commit 83b6cbcbdb33ae665afe90c4807d1a753b84b972
Author: = <sean.fioritto@gmail.com>
Date:   Mon Aug 12 15:51:22 2013 -0500

    Added doctype and font color.

commit 77bb1ef966daa82d07c010d98f7861a3a35ad156
Author: = <sean.fioritto@gmail.com>
Date:   Sun Aug 11 15:32:48 2013 -0500

    My first commit.
Macintosh-3:sketching sean$
```

Made some changes and committed them to the new branch.

Assuming you think your new changes are awesome, you want to merge them back into the master branch. You do this in two steps:

```
git checkout master
   git merge colors
```

The first step is to use git checkout master to switch back to the master branch. The second step merges the color branch changes into the master branch. Here's what this looks like:



```
Macintosh-3:sketching sean$ git checkout master
Switched to branch 'master'
Macintosh-3:sketching sean$ git log
commit 83b6cbcbdb33ae665afe90c4807d1a753b84b972
Author: = <sean.fioritto@gmail.com>
Date:   Mon Aug 12 15:51:22 2013 -0500

    Added doctype and font color.

commit 77bb1ef966daa82d07c010d98f7861a3a35ad156
Author: = <sean.fioritto@gmail.com>
Date:   Sun Aug 11 15:32:48 2013 -0500

    My first commit.
Macintosh-3:sketching sean$ git merge colors
Updating 83b6cbc..17a87e5
Fast-forward
 index.html |    7 ++++++-
 1 files changed, 6 insertions(+), 1 deletions(-)
Macintosh-3:sketching sean$ git log
commit 17a87e5f46387689792af9392919e9a260c23de9
Author: = <sean.fioritto@gmail.com>
Date:   Mon Aug 12 15:52:35 2013 -0500

    branches in git are awesome

commit 83b6cbcbdb33ae665afe90c4807d1a753b84b972
Author: = <sean.fioritto@gmail.com>
Date:   Mon Aug 12 15:51:22 2013 -0500

    Added doctype and font color.

commit 77bb1ef966daa82d07c010d98f7861a3a35ad156
Author: = <sean.fioritto@gmail.com>
Date:   Sun Aug 11 15:32:48 2013 -0500

    My first commit.
Macintosh-3:sketching sean$
```

Merging a branch into the master branch.

Check out the logs before the merge and after the merge. After the merge the commits you made in the colors branch are now in the log of the master branch. Now that everything is in the master branch and nothing went wrong you can delete the colors branch.

```
git branch -d colors
```

Keep it simple

You can make a branch on top of another branch. If you are in a new branch and run the checkout -b anotherBranch command, another-Branch will branch off from the curent branch instead of the master branch. Don't do this.

A branch should represent a new concept or direction; it should in-capsulate a big idea so you shouldn't be creating thousands of them. Git branches are amazing and incredibly powerful but if you don't exercise some restraint you will end up in the weeds quickly. One problem is merge conflicts, which you will learn how to handle, but they are not fun and are to be avoided.

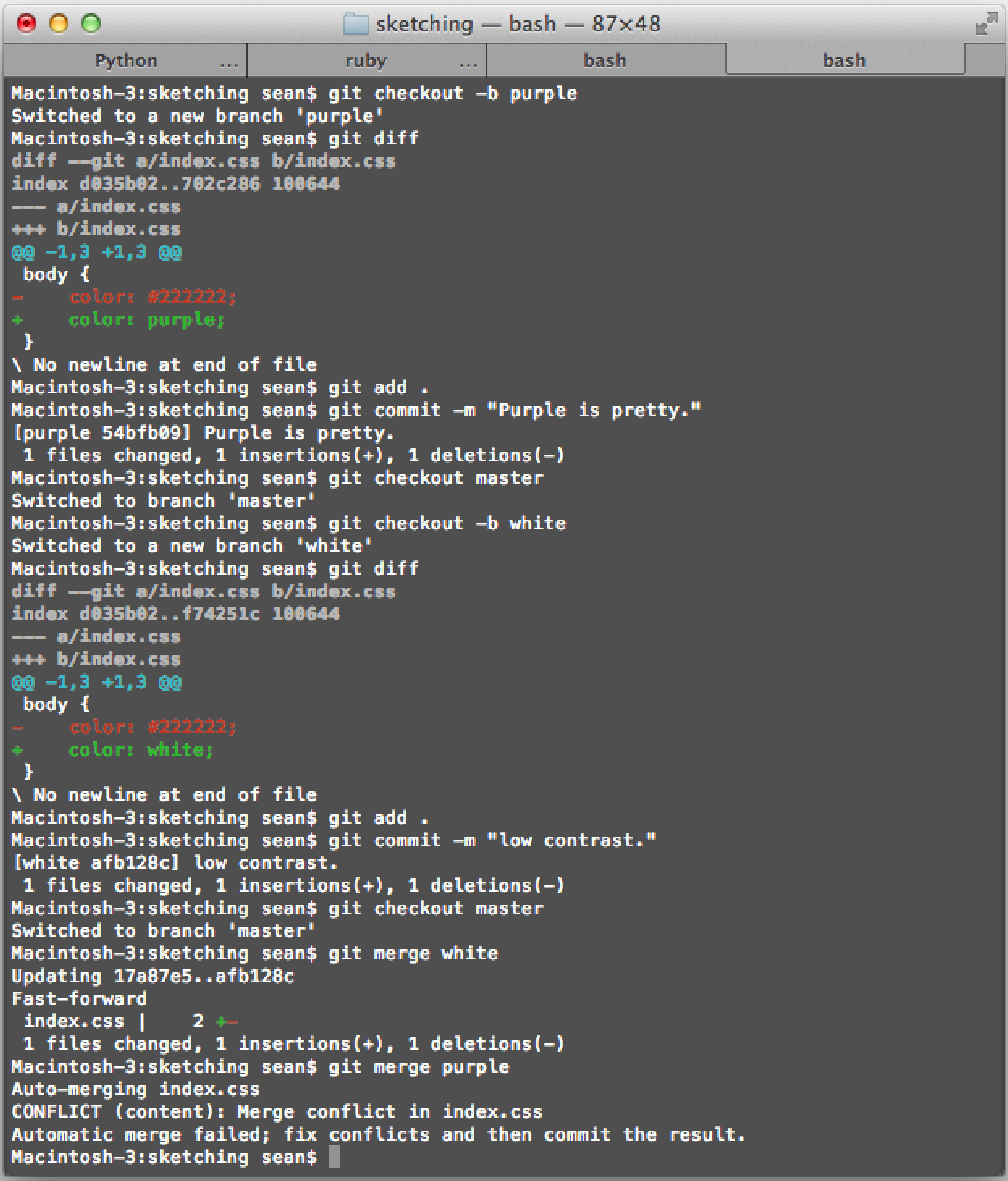
You could also make a new branch, do some work there, switch back to the master branch and do some more work, and then decide to merge the new branch into the master branch. Don't do this either. In-stead work on one branch at a time and then either keep those changes or discard them completely and start at the master branch.

Git supports extremely complicated workflows. You can do almost anything you can imagine. But unless you plan on spending countless hours mastering the hundreds of commands in Git not covered in this chapter, you should stick with the basics.

MERGE CONFLICT

When working with branches you can have something called a merge conflict. Merge conflicts can be troublesome, but the good news is if you follow the workflow recommended in this chapter, merge conflicts will be rare. Also, for the most part fixing a merge conflict is not terribly difficult but if they happen often, it can become a waste of time, so avoid them by keeping your workflow simple.

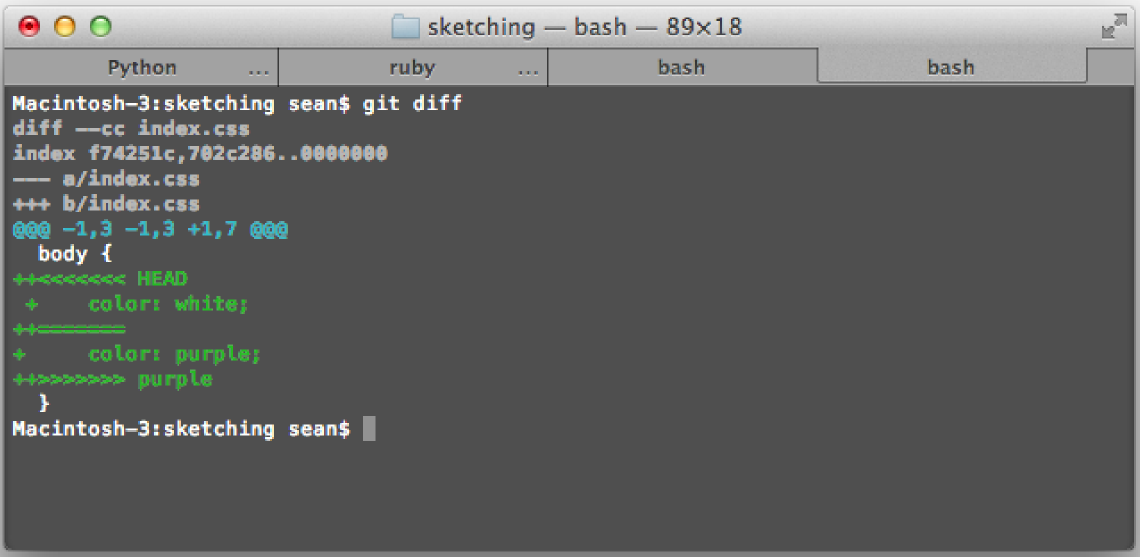
How does a merge conflict happen? Imagine you are working on one branch and you change the default font color to be purple. On a whim you create another branch and make the default font color white, just to mess with your client. So you merge the white branch and then later go to merge the purple branch — the problem is you have changed the same line in the same file in two different branches and Git has no idea what to do. The merge conflict in this case is Git asking you, "Do you want purple or white?" This is what it looks like when you have a merge conflict:



```
Macintosh-3:sketching sean$ git checkout -b purple
Switched to a new branch 'purple'
Macintosh-3:sketching sean$ git diff
diff --git a/index.css b/index.css
index d035b02..702c286 100644
--- a/index.css
+++ b/index.css
@@ -1,3 +1,3 @@
 body {
-    color: #222222;
+    color: purple;
 }
\ No newline at end of file
Macintosh-3:sketching sean$ git add .
Macintosh-3:sketching sean$ git commit -m "Purple is pretty."
[purple 54bfb09] Purple is pretty.
 1 files changed, 1 insertions(+), 1 deletions(-)
Macintosh-3:sketching sean$ git checkout master
Switched to branch 'master'
Macintosh-3:sketching sean$ git checkout -b white
Switched to a new branch 'white'
Macintosh-3:sketching sean$ git diff
diff --git a/index.css b/index.css
index d035b02..f74251c 100644
--- a/index.css
+++ b/index.css
@@ -1,3 +1,3 @@
 body {
-    color: #222222;
+    color: white;
 }
\ No newline at end of file
Macintosh-3:sketching sean$ git add .
Macintosh-3:sketching sean$ git commit -m "low contrast."
[white afb128c] low contrast.
 1 files changed, 1 insertions(+), 1 deletions(-)
Macintosh-3:sketching sean$ git checkout master
Switched to branch 'master'
Macintosh-3:sketching sean$ git merge white
Updating 17a87e5..afb128c
Fast-forward
 index.css |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
Macintosh-3:sketching sean$ git merge purple
Auto-merging index.css
CONFLICT (content): Merge conflict in index.css
Automatic merge failed; fix conflicts and then commit the result.
Macintosh-3:sketching sean$ 
```

A Git session that ends in a merge conflict.

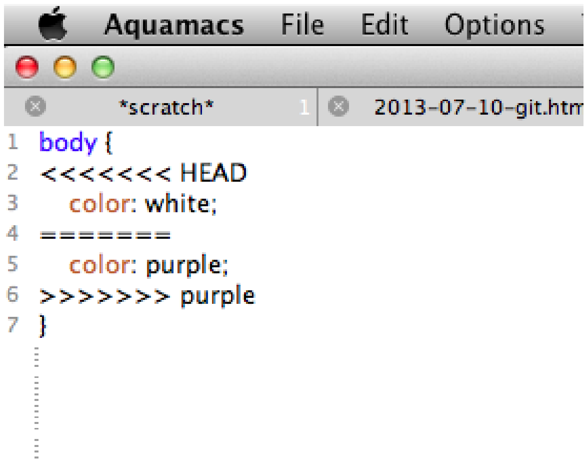
The scenario is a bit contrived, and it probably won't happen when you are working by yourself, but if it does this is how to fix it. Immediately after a merge conflict Git drops some special markings into your files to show you where the problems were. If you do git diff at this point you will see exactly where those markings are:



```
Macintosh-3:sketching sean$ git diff
diff --cc index.css
index f74251c,702c286..0000000
--- a/index.css
+++ b/index.css
@@@ -1,3 -1,3 +1,7 @@@
  body {
++<<<<<<< HEAD
 +    color: white;
++=======
+     color: purple;
++>>>>>>> purple
  }
Macintosh-3:sketching sean$ 
```

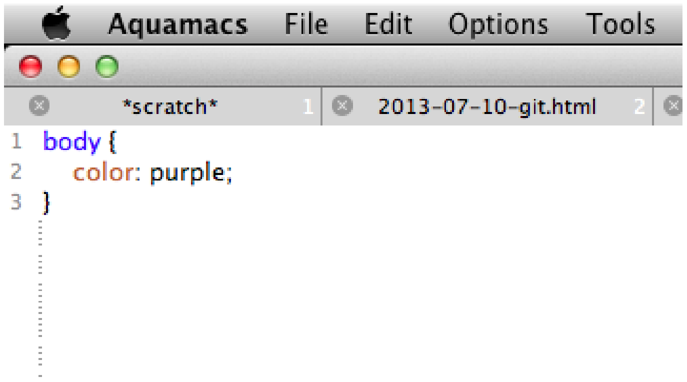
Git leaves special marks in files where there were merge conflicts.

If you were a little confused up to this point these markings might help you understand what's going on. You can see Git knows you made changes to the color of the font in two different branches, but it doesn't know which one you want to keep. It can't assume one or the other, so it leaves it up to you. All you have to do is go into the file, edit it to reflect the change you want, save it, and commit the change.



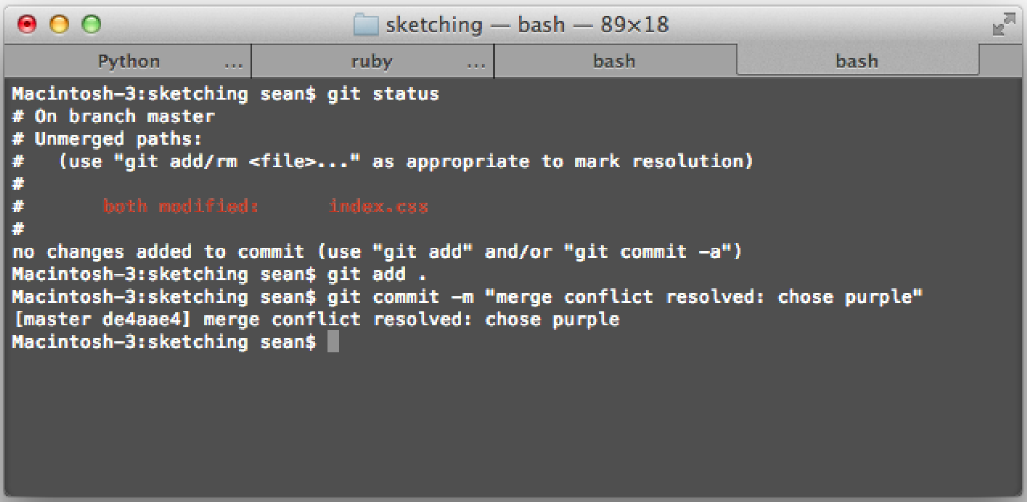
```
body {
<<<<<<< HEAD
   color: white;
=======
   color: purple;
>>>>>>> purple
}
```

Should the color be white or purple?



```
body {
   color: purple;
}
```

If the client wants purple...



```
Macintosh-3:sketching sean$ git status
# On branch master
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:      index.css
#
no changes added to commit (use "git add" and/or "git commit -a")
Macintosh-3:sketching sean$ git add .
Macintosh-3:sketching sean$ git commit -m "merge conflict resolved: chose purple"
[master de4aae4] merge conflict resolved: chose purple
Macintosh-3:sketching sean$ 
```

When you are done making changes, add them and commit them just like normal.

When everything else fails

If you are keeping it simple and using branches to try out new ideas, most of the time Git will just work. But you will inevitably run into problems, and when you do, git reset can sometimes bail you out. Before you dive into git reset, a word of warning: some of these commands are hard to undo — redo in Git is not as easy as undo! Always do a git status to make sure you know exactly what state your repository is in before executing any of these commands. And if you are working with a team and sharing a central repository, (you'll know if you are), *don't use any of these commands*. Some of them are fine to use, but better safe than sorry.

If you really want to be sure you don't lose data, create a copy of your repository. If you use Github, (you'll learn about Github later at the end of this chapter), then you will always have an extra copy of all of your work and you don't need to worry too much about losing data. But if you're not using Github, a quick workaround is to just copy/paste your entire repository to another folder before you do anything risky. You can also use a command called git clone.

```
git clone ~/sketching ~/sketching-clone
```

... or if you are on windows ...

```
git clone c:/sketching c:/sketching-clone
```

If you run this command it will create a complete copy of your repository in the ~/sketching-clone folder. If something goes wrong, just rename it to sketching and you are back in business.

With a backup in place, you are ready to use the git reset command. As you create commits, Git keeps track of each change in the log. Each commit is a specific point in time you can jump back to, and git reset is basically time travel for your code. Want to reset everything to the commit you made a few hours ago? No problem, git reset can do that for you.

Here are a few scenarios you may run into and how to use git reset to get out of trouble.

- You tried something out you didn't like and you haven't yet committed the changes. In other words, if you run git status you will see all of the files you changed in the "Changes not staged for commit" section. How do you get rid of these changes and get all of your files back to exactly where you started?

  ```
  git reset --hard head
  ```

  Warning: the --hard option deletes your changes. You can't get them back.

- You made a mistake in your last commit. Maybe you added some files you didn't want to commit, or you found a typo. Most of the time it makes more sense to just keep the commit and put any fixes in a new commit, but on occasion you may want to undo the commit and just try again.

  ```
  git reset head~
  ```

  If you run git status you will see everything you just committed is now unstaged and if you run git log you will see your last commit has been deleted.

- You made a change and commited it. The problem is, you liked what you did but you changed your mind: now you hate it and there is no chance you want to keep it. You can completely delete a commit and, in essence, take your repository back to the last good commit you liked.

  ```
  git reset --hard head~
  ```

  The --hard option ensures the changes are completely deleted, (unlike in the previous scenario, you don't want your changes to stick around). The head~ part of the command tells git to go back to the previous commit, head~~ goes back two commits, etc.

- You've just done a hard reset of your repository, but you made a mistake and you need to get the code back to where it was. In other words, you need to undo your last use of git reset.If you've made other commits since undoing, you're out of luck. Hopefully you made a backup or have a clone sitting around.

  ```
  git reset head@{1}
  ```

Git'n Advanced

Git is a state-of-the-art, distributed version control system lovingly, handcrafted in pure C by Linus Torvalds, computer wizard, hero of open source and inventor of Linux. Torvalds wrote Git to manage the 3,500 lines of code merged into Linux Kernel every, single, day. Needless to say, in this chapter you have barely scratched the surface of what you can do with this tool. What you have learned is enough to get started, but if you want to learn more about Git here are some articles and books worth reading and advanced topics to look into.

Github

Git is designed for collaboration. Each repository can be cloned and it is very easy to take any commits you have made and send them to another clone of your repository. It's so easy to do this, in fact, that it is rare to see a standalone Git repository without a clone living on a server somewhere and most of the time when you start a project with Git you start by cloning a repository from a server.

Github is the defacto web service for creating and storing clones of Git repositories on a remote server. It comes loaded with all kinds of tools for collaboration and it's free for any open source project. Github is so popular it's even common for people to say Github instead of Git — to many, they are inseparable.

Over the years Github has created excellent documentation for Git and most of it lives at help.github.com. Here are a few selections you might find helpful or interesting:

- Getting set up with Git
- Creating a new repository on Github
- Ignoring files in your repository
- Screencasts for Git basics
- A huge list of resources for Git on the web

Books and articles

If you run out of reading on the Github help page, or if it seems overwhelming, here are a few hand picked options for advanced reading on Git.

- Mastering Git Basics — Github founder, Tom Preston-Warner takes some of the voodoo out of Git in this excellent presentation
- Pro Git is an open source, (read, free), book all about Git.
- Try Git is an interactive, web application for learning Git basics. You type commands in as you go through the tutorial.
- Git Immersion — "A guided tour that walks through the fundamentals of git, inspired by the premise that to know a thing is to do it."
- Version Control with Git — The O'Reilly Git book.