Intro to Source Control

Nick McSpadden
Client Platform Engineer, Facebook

Intro to `git`

Nick McSpadden
Client Platform Engineer, Facebook

What does "source control" mean?

Imagine collaboratively editing a Microsoft Word document 15 years ago...

Sharing a Word Document, 2003

File.doc

Sharing a Word Document, 2003

File.doc

Sharing a Word Document, 2003

File.doc

File.doc

Sharing a Word Document, 2003

File2.doc

File.doc

Sharing a Word Document, 2003

File2.doc

File.doc

They no longer have the same contents,

has an old version

Sharing a Word Document, 2003

File2.doc

File2.doc

They've both made independent changes...

Sharing a Word Document, 2003

File2.doc

File2.doc

How do you combine these back into one document?

Sharing a Word Document, 2003

File2.doc

File2.doc

How do you combine these back into one document?

Collaborative editing of things like simple documents was already a pain.

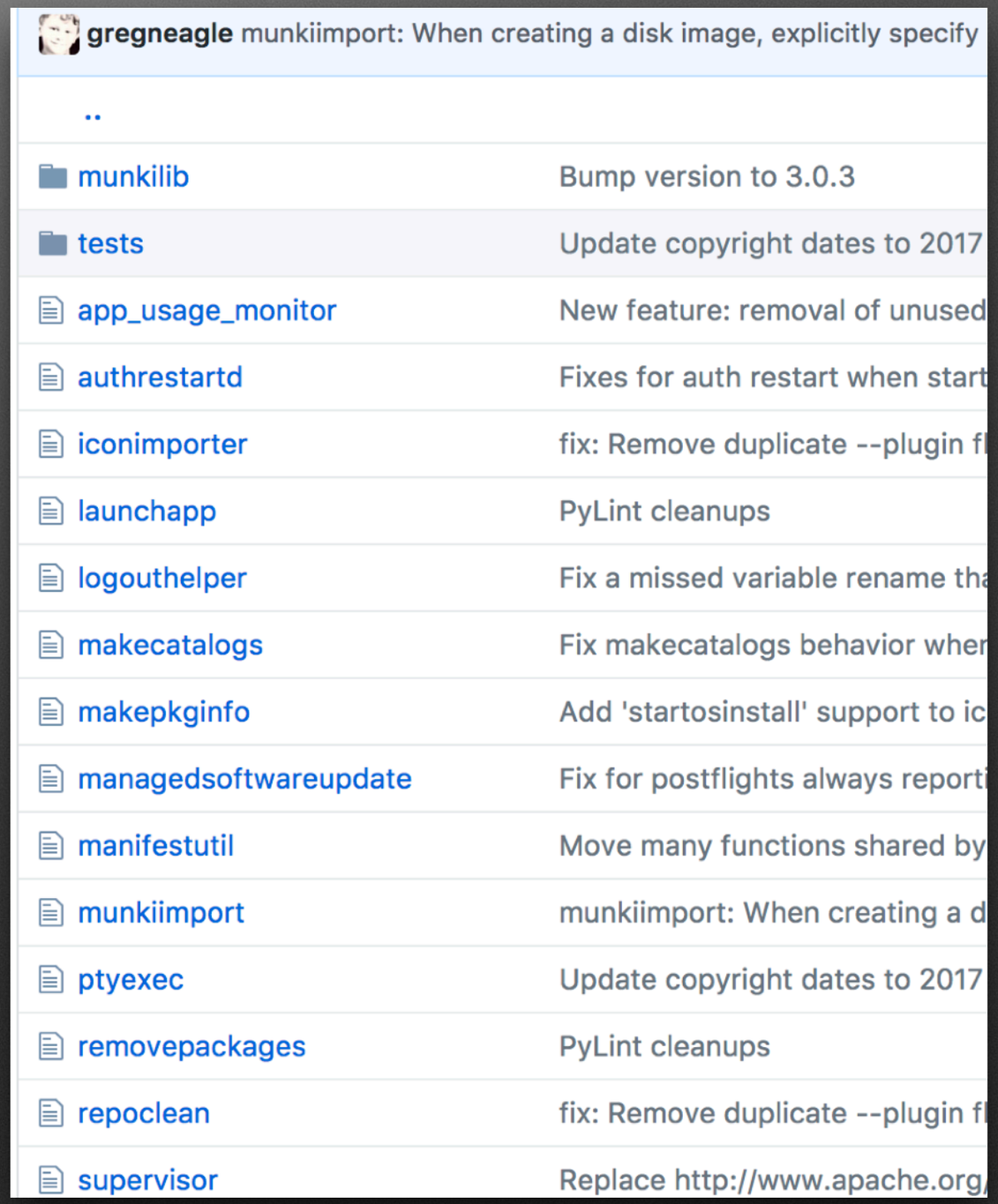
Imagine tying to share something much more complex and extensive, like the source code to a large piece of software?

Collaborative editing of things like simple documents was already a pain.

Imagine tying to share something much more complex and extensive, like the source code to a large piece of software?

Collaborating on projects

- Programming projects can involve large numbers of files across large trees of directories
- You generally need all of these files to work on the project
- We need to track edits made by anyone to any of these files



"Source control" is a mechanism to track revisions made to programming projects.

Source Control

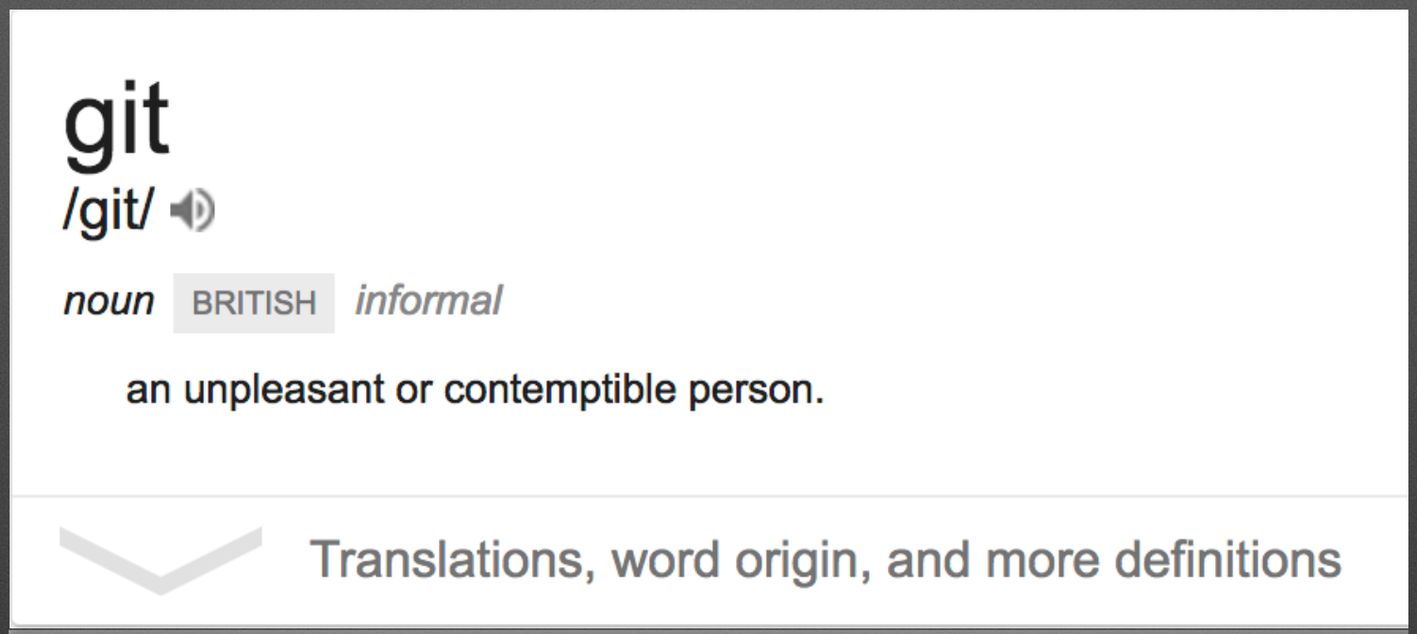
- Everyone has all of the code.
- Everyone has the ability to share their edits back.
- Everyone has the ability to intelligently absorb other contributions.

Source Control

- http://lockergnome.com/2005/07/25/an-introduction-to-basic-source-control-principles/
- https://dzone.com/articles/10-commandments-good-source
- https://www.lynda.com/Visual-Studio-tutorials/Principles-source-control/487943/513779-4.html#tab
- https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

What is `git`?

Well, Google's on point...



Lots of good resources!

What is Git: become a pro at Git with this guide | Atlassian Git Tutorial
https://www.atlassian.com/git/tutorials/what-is-git ▾
By far, the most widely used modern version control system in the world today is **Git**. **Git** is a mature, actively maintained open source project originally developed in 2005 by Linus Torvalds, the famous creator of the Linux operating system kernel.

Git - Git Basics
https://git-scm.com/book/en/Getting-Started-Git-Basics ▾
Git doesn't think of or store its data this way. Instead, **Git** thinks of its data more like a set of snapshots of a miniature filesystem. Every time you commit, or save the state of your project in **Git**, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.

Git Basics Episode 2
https://git-scm.com/video/what-is-git ▾
The entire Pro **Git** book written by Scott Chacon and Ben Straub is available to read online for free. Dead tree versions are ... **Git** Basics Episode 2. **What is Git**?

Git - Wikipedia
https://en.wikipedia.org/wiki/Git ▾
Git is a version control system (VCS) for tracking changes in computer files and coordinating work on those files among multiple people. It is primarily used for ...

Developer(s): Junio Hamano and others
Operating system: POSIX: Linux, Windows, macOS
Initial release: 7 April 2005; 12 years ago
Stable release: 2.13.2 / 25 June 2017; 2 days ago

What is git? | Opensource.com
https://opensource.com/resources/what-is-git ▾
Jul 7, 2016 - Welcome to my series on learning how to use the **Git** version control system! In this introduction to the series, you will learn what **Git** is for and ...

What is Git? | Learn Git - Visual Studio
https://www.visualstudio.com/learn/what-is-git/ ▾
Apr 4, 2017 - **Git** is the most commonly used version control system today. Will it be the standard for the future?

What is `git`?

What is Git?

By Kayla Ngan

Git is the most commonly used version control system today and is quickly becoming *the* standard for version control. Git is a distributed version control system, meaning your local copy of code is a complete version control repository. These fully-functional local repositories make it is easy to work offline or remotely. You commit your work locally, and then sync your copy of the repository with the copy on the server. This paradigm differs from centralized version control where clients must synchronize code with a server before creating new versions of code.

https://www.visualstudio.com/learn/what-is-git/

`git` is one of many tools for source control

Principles of `git`

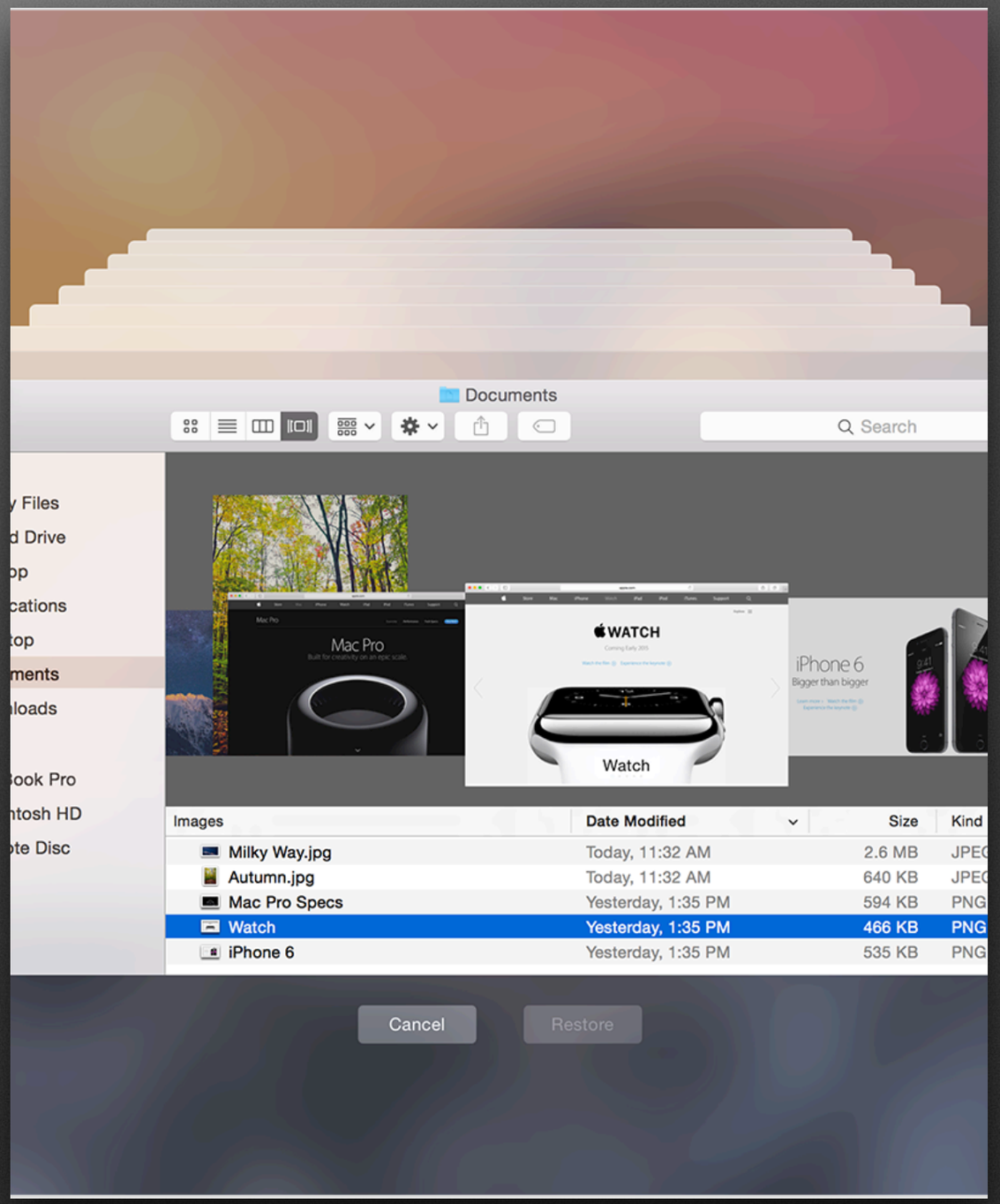
- Everyone has a complete copy of the repo, which includes all files, all revisions, and all *history* of all revisions
- There is one `master` branch which is the Source of Truth for the project
- All changes made are stored in history forever; you generally only ever add data to the repo
- Each revision is a snapshot of the entire project

Metaphors for `git`

- Time Machine is a good example of the concept of snapshot-based history

- If a file hasn't changed, Time Machine just 'notes' that it's the same as the original

- You can go back in time to look at a complete picture of your drive



Metaphors for `git`

- Wikipedia is a good example of recording history of changes
- Every time an edit is made, it's recorded in history, even if that change is later reverted
- You can go back in time to look at at that page at any point in its history

Apple Inc.: Revision history

View logs for this page

Search for revisions

From year (and earlier): 2017 From month (and earlier): all Tag f

For any version listed below, click on its date to view it. For more help, see Help:Page history a
External tools: Revision history statistics · Revision history search · Edits by user · Numb

(cur) = difference from current version, (prev) = difference from preceding version, **m** = minor e
(newest | oldest) View (newer 50 | older 50) (20 | 50 | 100 | 250 | 500)

Compare selected revisions

- (cur | prev) 06:04, 4 July 2017 LocalNet (talk | contribs) **m** . . (240,330 bytes) (+3) . .
- (cur | prev) 21:29, 3 July 2017 Frietjes (talk | contribs) . . (240,327 bytes) (-1) . . *(Cle*
- (cur | prev) 06:54, 3 July 2017 Adrian J. Hunter (talk | contribs) **m** . . (240,328 bytes)
- (cur | prev) 06:43, 3 July 2017 Ylee (talk | contribs) . . (240,250 bytes) (+448) . . *(→E*
- (cur | prev) 08:30, 2 July 2017 LocalNet (talk | contribs) . . (239,802 bytes) (0) . . *(As*

  it's finally correct and up-to-date :P)
- (cur | prev) 07:19, 1 July 2017 LocalNet (talk | contribs) . . (239,802 bytes) (-1) . . *(Ta*
- (cur | prev) 17:38, 30 June 2017 Bender the Bot (talk | contribs) **m** . . (239,803 bytes
- (cur | prev) 12:11, 30 June 2017 LocalNet (talk | contribs) . . (239,797 bytes) (+272)
- (cur | prev) 11:26, 30 June 2017 LocalNet (talk | contribs) . . (239,525 bytes) (-379) .
- (cur | prev) 11:19, 30 June 2017 LocalNet (talk | contribs) . . (239,904 bytes) **(-1,031)**

  source. A Tokyo, Japan store had been opened in 2003; this concerned a second store)
- (cur | prev) 11:14, 30 June 2017 LocalNet (talk | contribs) . . (240,935 bytes) **(+602)**
- (cur | prev) 11:04, 30 June 2017 LocalNet (talk | contribs) . . (240,333 bytes) (0) . . *(I*
- (cur | prev) 11:01, 30 June 2017 LocalNet (talk | contribs) . . (240,333 bytes) (0) . . *(I*
- (cur | prev) 10:33, 30 June 2017 LocalNet (talk | contribs) . . (240,333 bytes) **(-533)** .
- (cur | prev) 18:14, 27 June 2017 KAP03 (talk | contribs) . . (240,866 bytes) (+328) . .
- (cur | prev) 18:29, 25 June 2017 LocalNet (talk | contribs) . . (240,538 bytes) **(-671)**

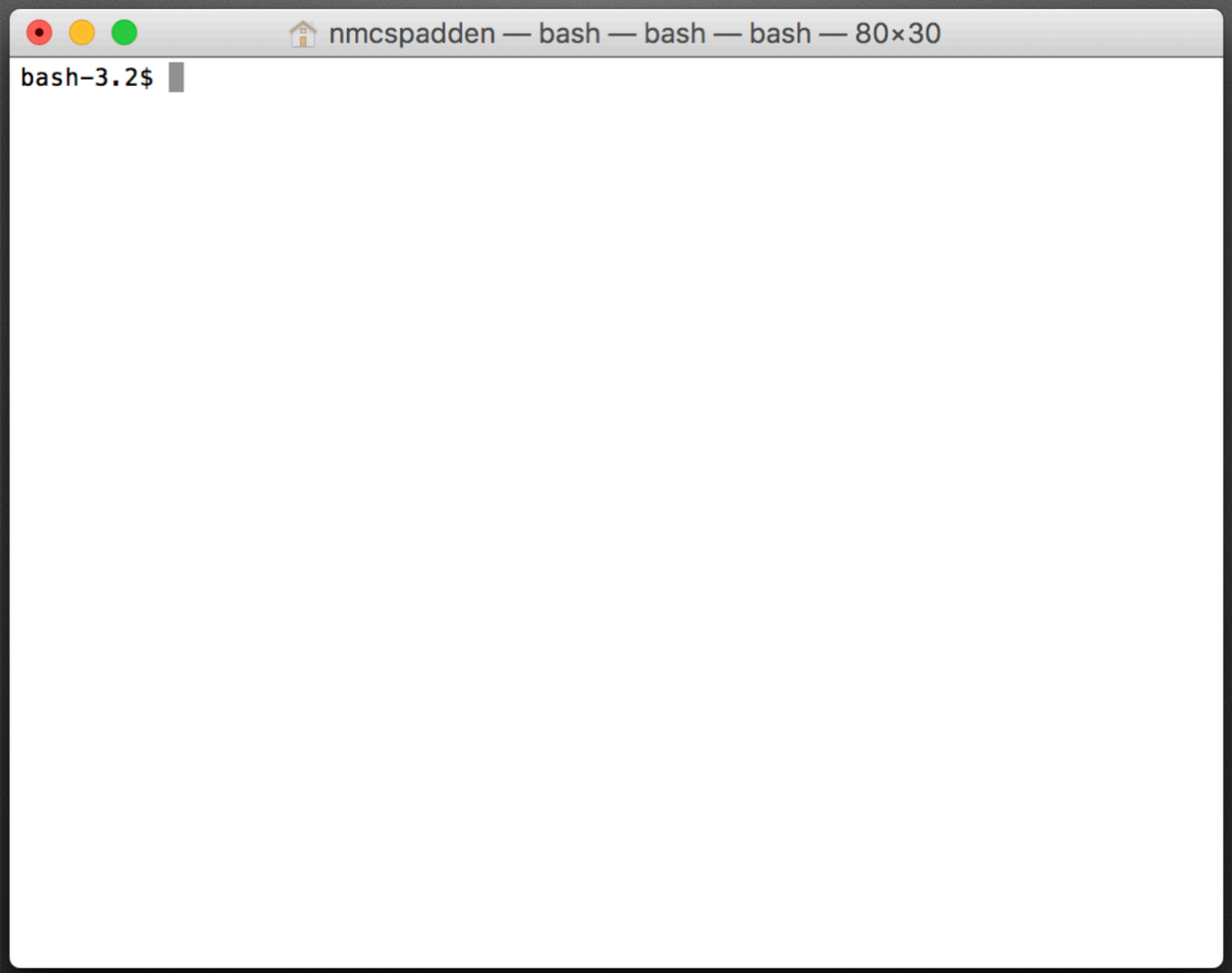
We want to store change history like Wikipedia...
We want to store snapshots like Time Machine...
We also want the ability for others to do the same thing!

Enough theory, let's get to practical

Common `git` terms

- `repo` - the "store" of all the different pieces of the project.
- `commit` - a saved revision to something inside the project
- `master` - the Source of Truth of the state of all files in the project
- `branch` - a deviation from `master` so you can pursue multiple different ideas at once

Getting started with git



It starts with /Applications/Utilities/Terminal.app

Getting started with `git`

Installing git on macOS is easy:

```
$ xcode-select --install
```

https://help.github.com/articles/set-up-git/

Getting started with `git`

Check your version:

```
$ git --version

git version 2.11.0 (Apple Git-81)
```

Configure your email and user name for commits:

```
$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com
```

Getting started with `git`

Use a good text editor!

- Atom (free)
- TextMate (free)
- SublimeText (free with nag for paid version)
- BBEdit (free with nag for paid version)

Our first `git` project

Create our first project:

```
$ mkdir -p ~/code/
MyFirstGit

$ cd ~/code/MyFirstGit

$ git init
```

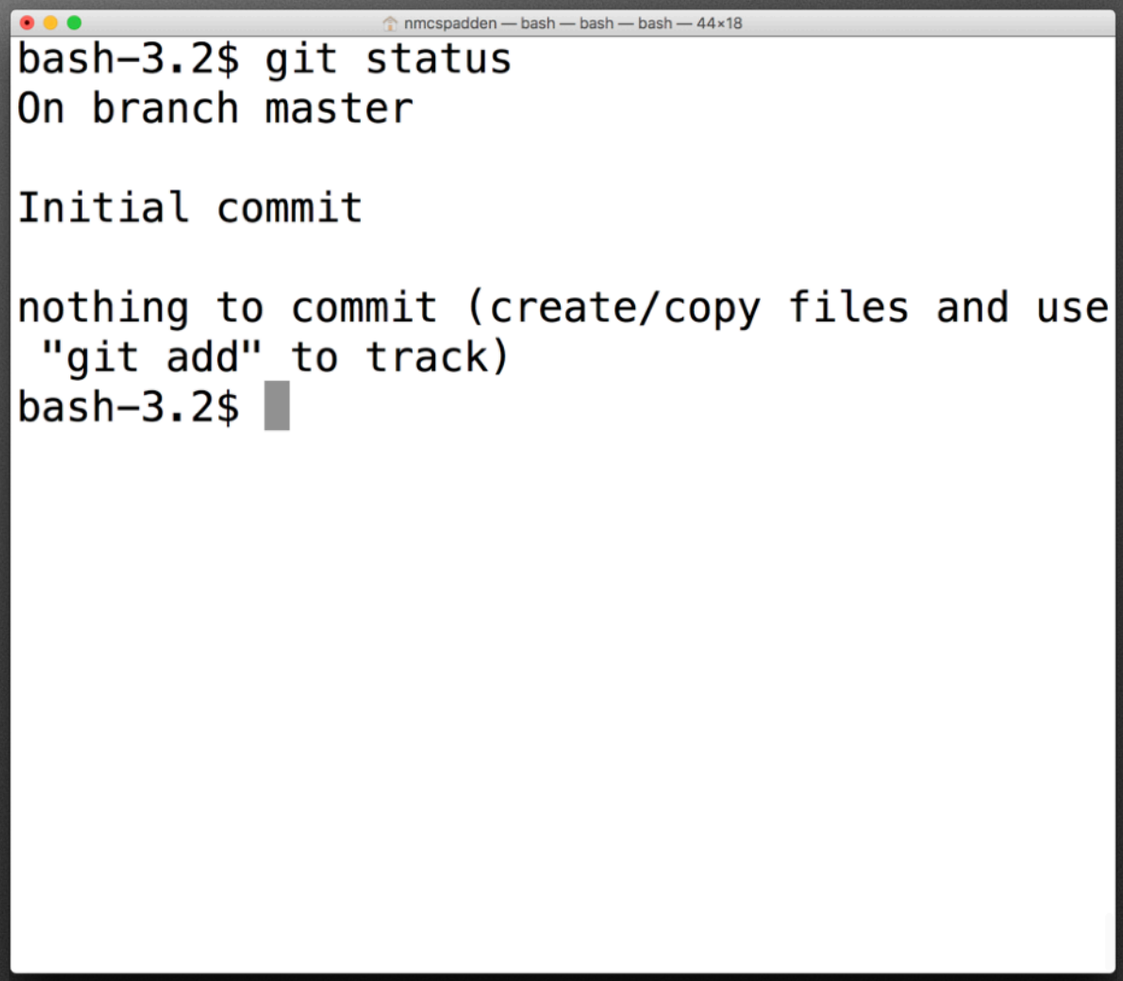
`git init` creates a new git repository in the current directory.

Our first `git` project

What's going on in our project?

```
$ git status
```

Branch: `master`
Commit: `Initial`
State: `clean`

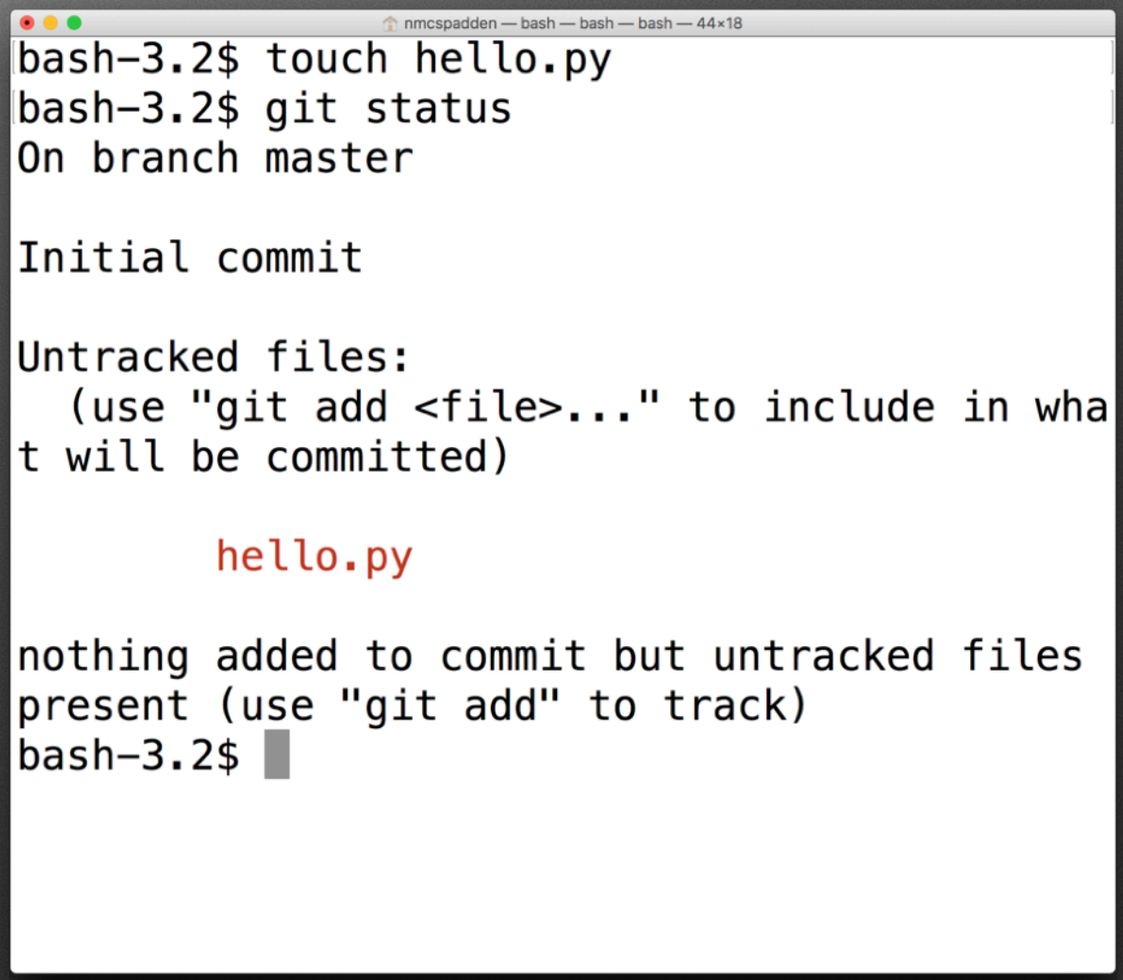


Our first `git` commit

Let's create a file and add it!

```
$ touch hello.py

$ git status
```

`git status` tells us the current state of the repo is.

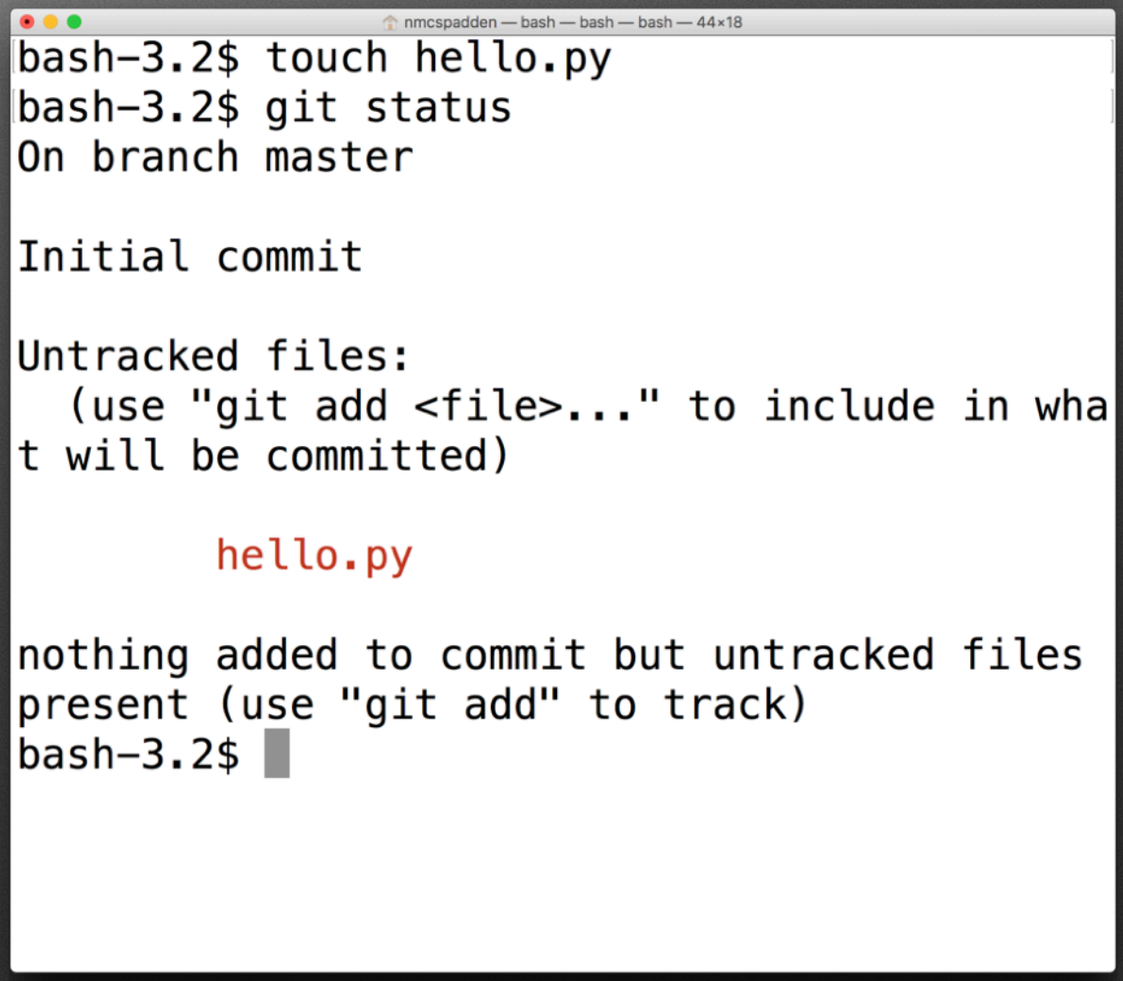


Our first `git` commit

"Untracked files" - a list of files that `git` is not aware of, but are present in the project directory tree

The `git` repo is currently "dirty": there are changes present that haven't been committed.

Any time you save changes to a file inside the repo, you will make the repo dirty.



Our first `git` commit

We want to make a new commit with this file, which means we must first `add` it to the staging area.

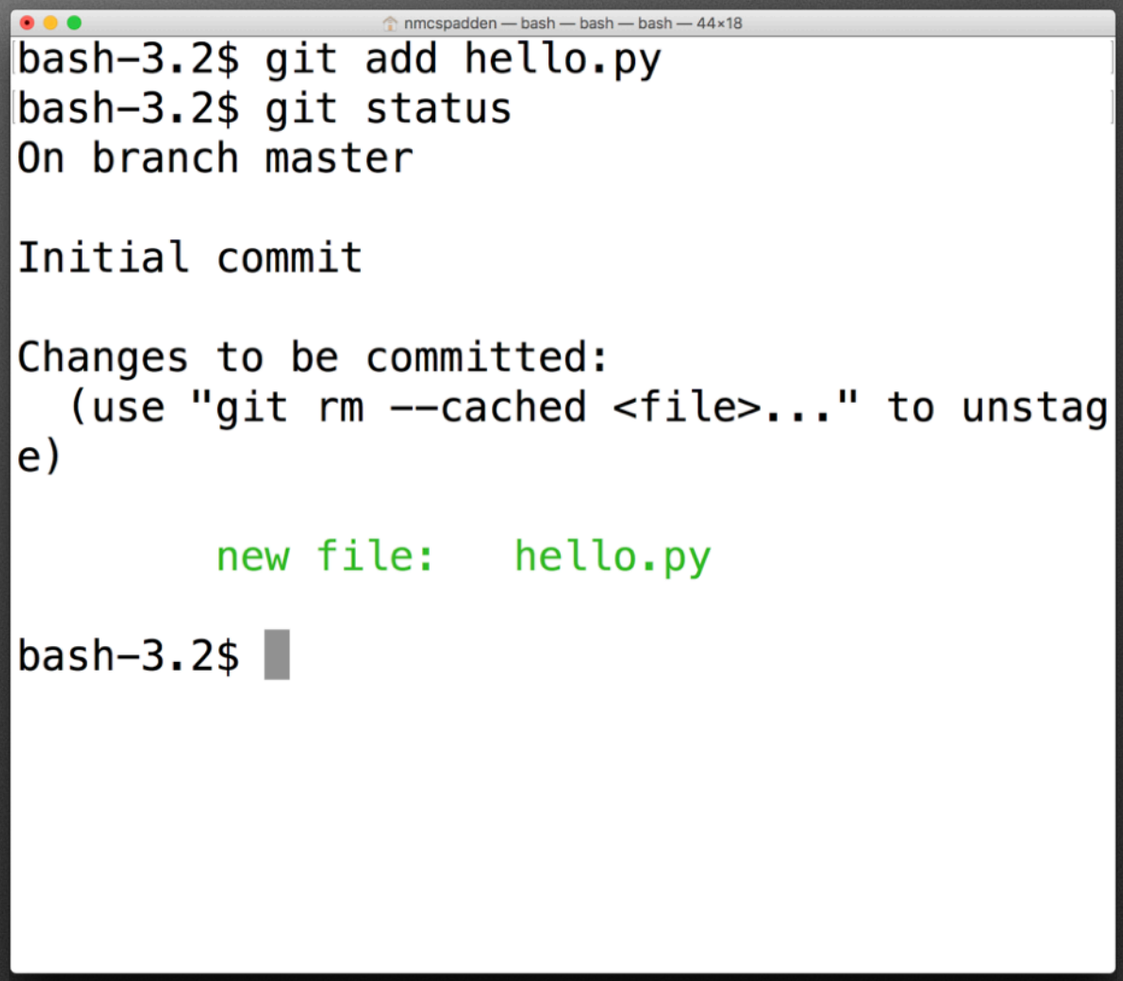
```
$ git add hello.py
```

or

```
$ git add --all
```

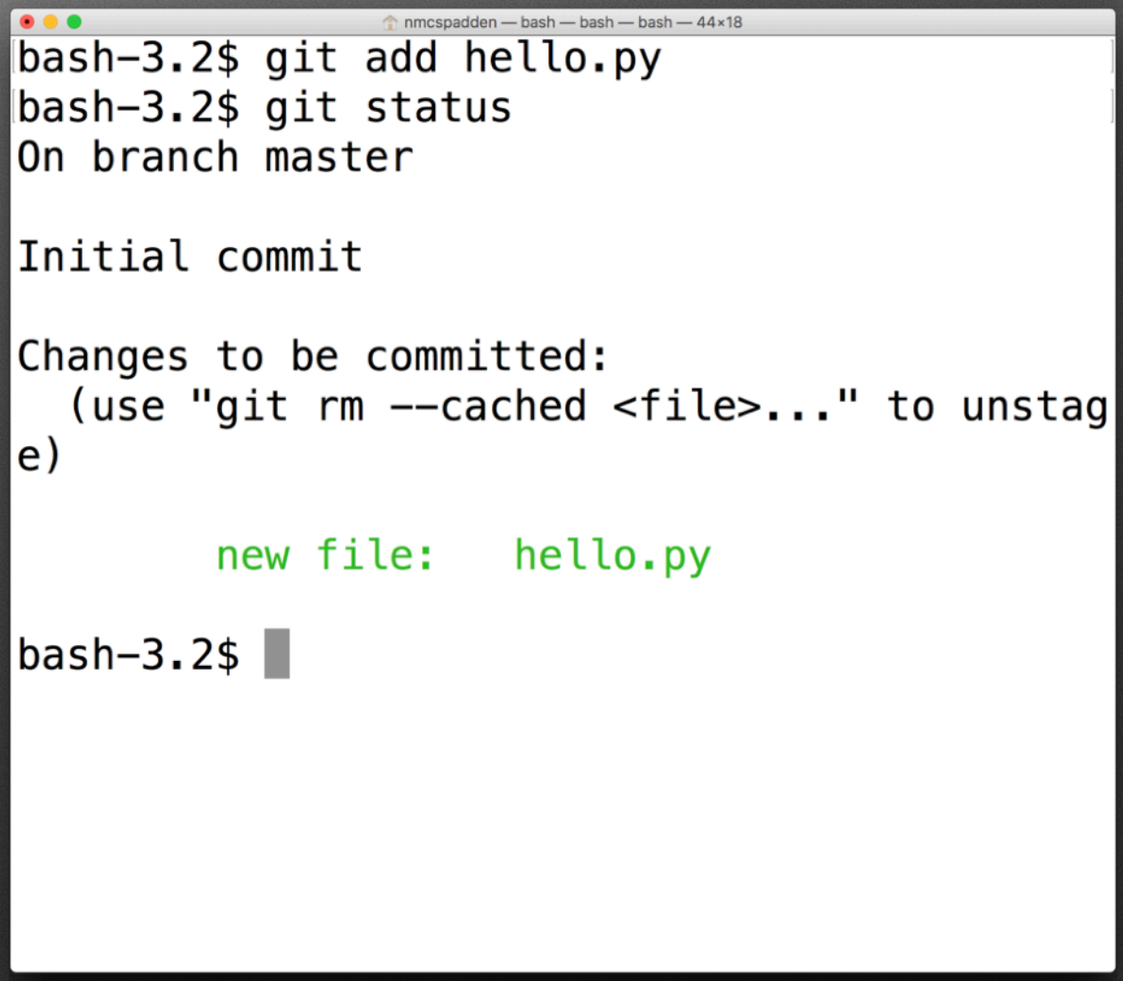
or

```
$ git add -A
```



Our first `git` commit

Once a file has been added, it's in the staging area. Items in the staging area can be committed.

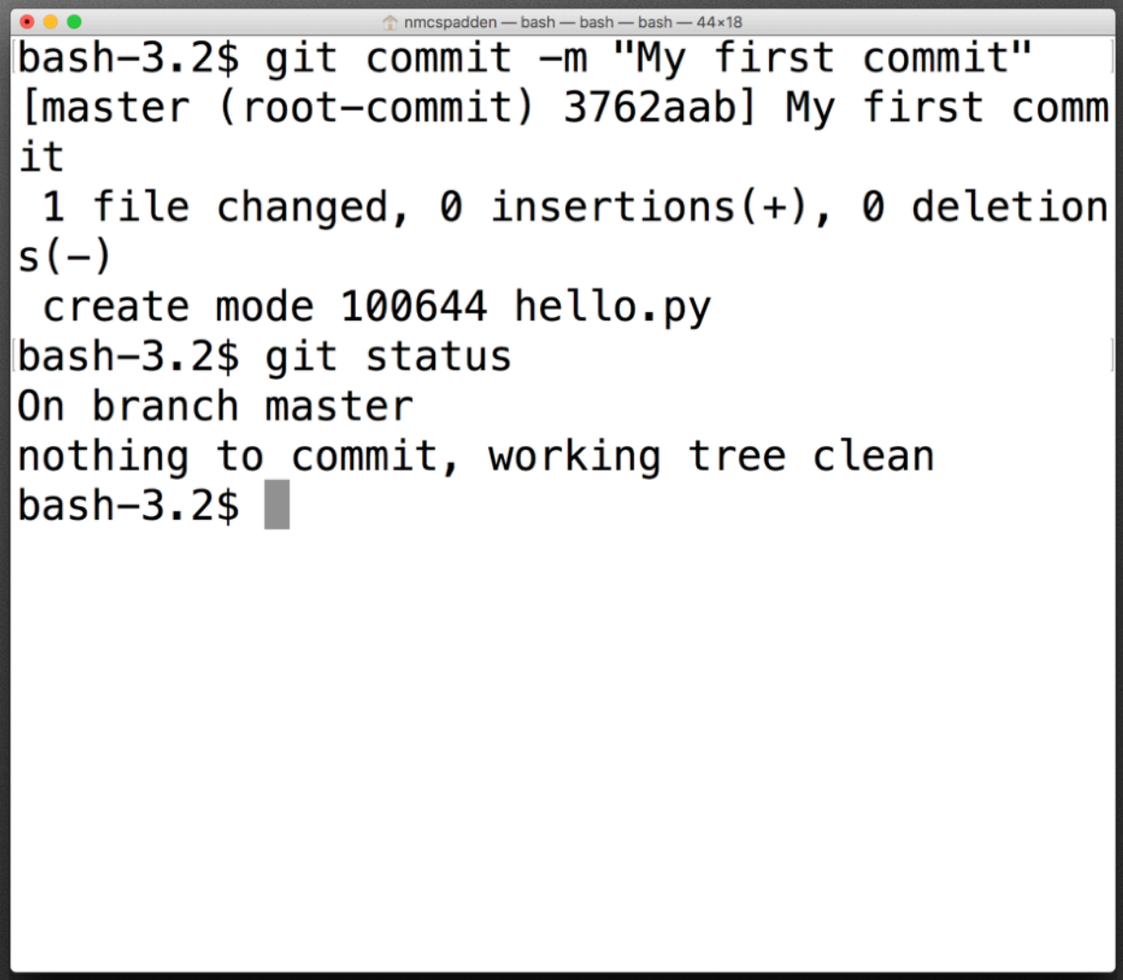


Our first `git` commit

Now we can go ahead and commit the change:

```
$ git commit -m "My first commit"
```

The `-m` argument means "Use this commit message". All commits must have a (brief) message indicating what the change is.



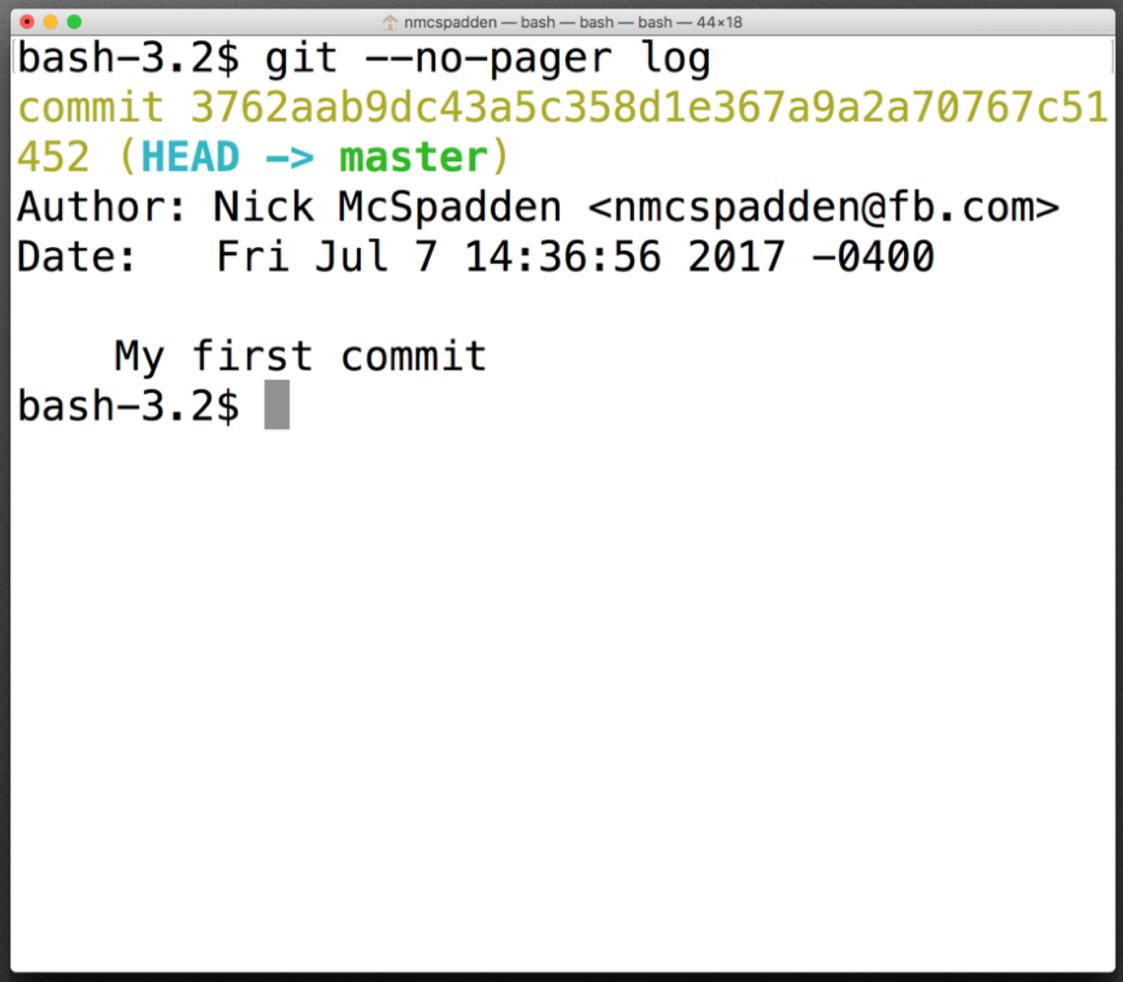
Our first `git` commit

How do we see the commit history?

```
$ git --no-pager log
```

This shows all commits made to the repo.

`--no-pager` makes it print the contents out to the terminal, rather than a page viewer.



We now have our first commit!

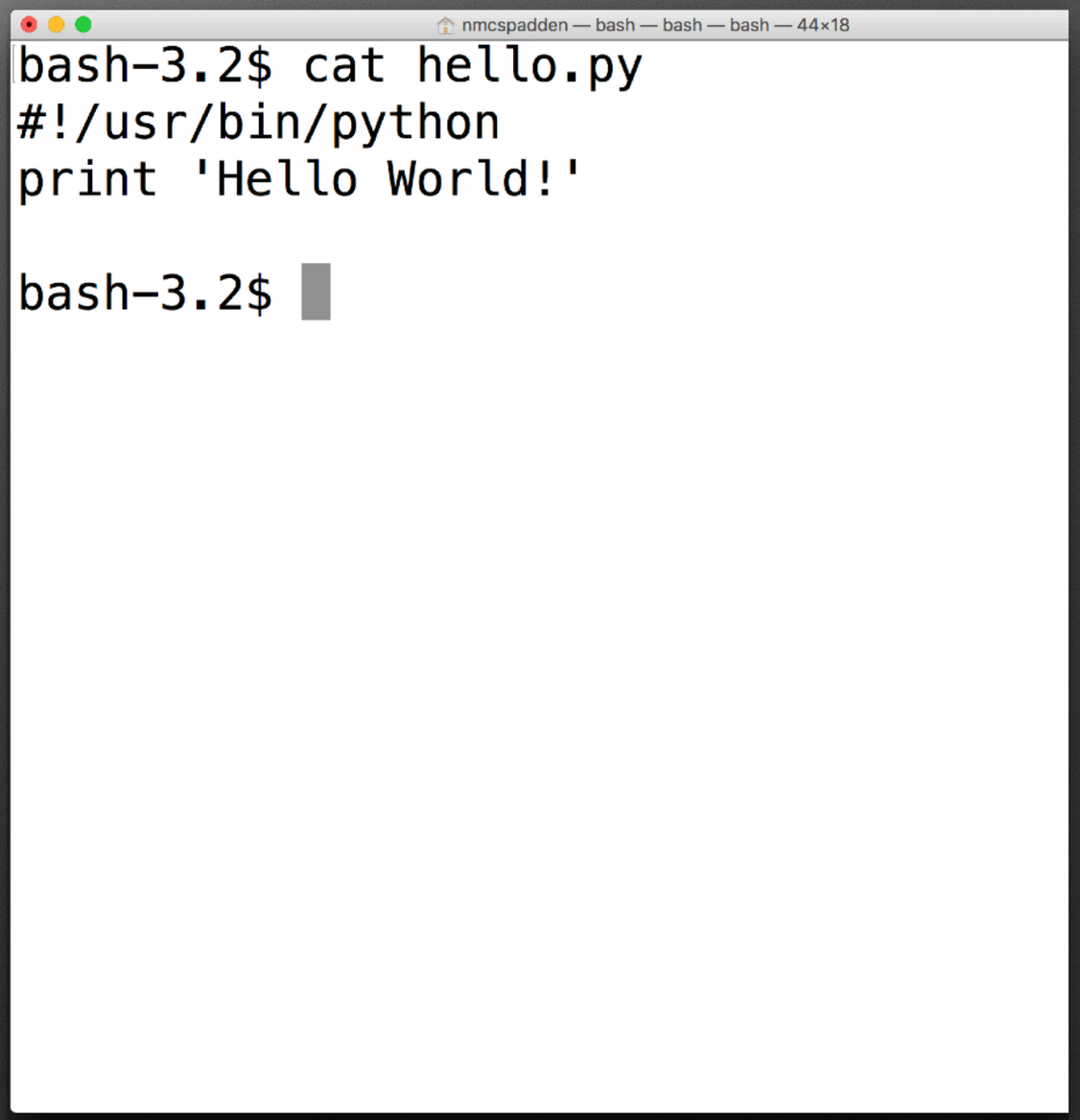
Unfortunately, it only goes downhill from here.

Let's add some real content

- No programming skill required!

hello.py is currently an empty file. Let's put actual code into it. Open up hello.py in a text editor:

```
#!/usr/bin/python
print 'Hello world!'
```

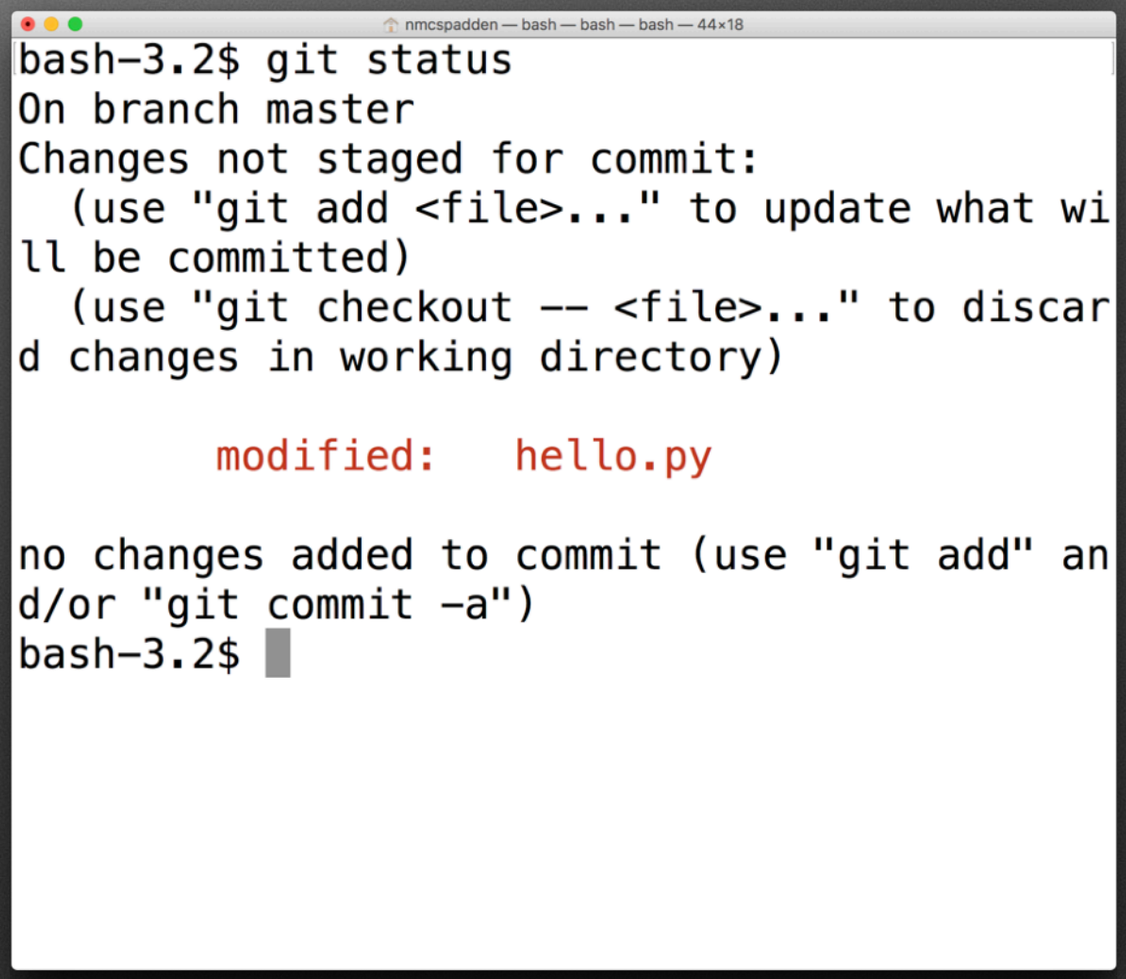


Let's add some real content

`git` tries very hard to tell you all of your options.

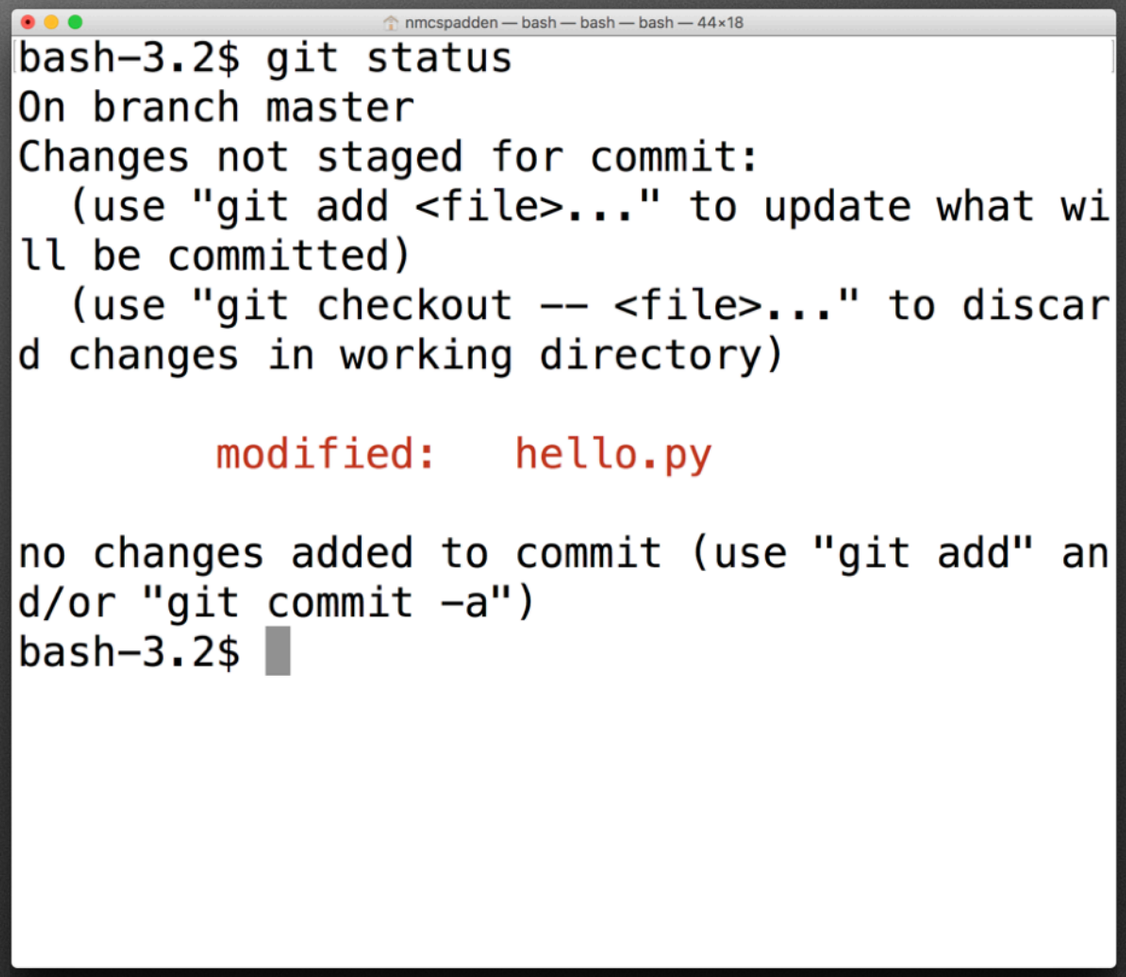
With a change made to a tracked file, it's telling us we can:

- `git add` it to the next commit, and proceed forward
- `git checkout` to discard the changes we made and go back to where we just were



Let's add some real content

Right now, we have unstaged changes waiting. The repo is currently dirty and we must decide how to proceed.



Let's add some real content

Let's commit our changes!

```
$ git add -A

$ git commit -m "Hello
World now runs"
```

or

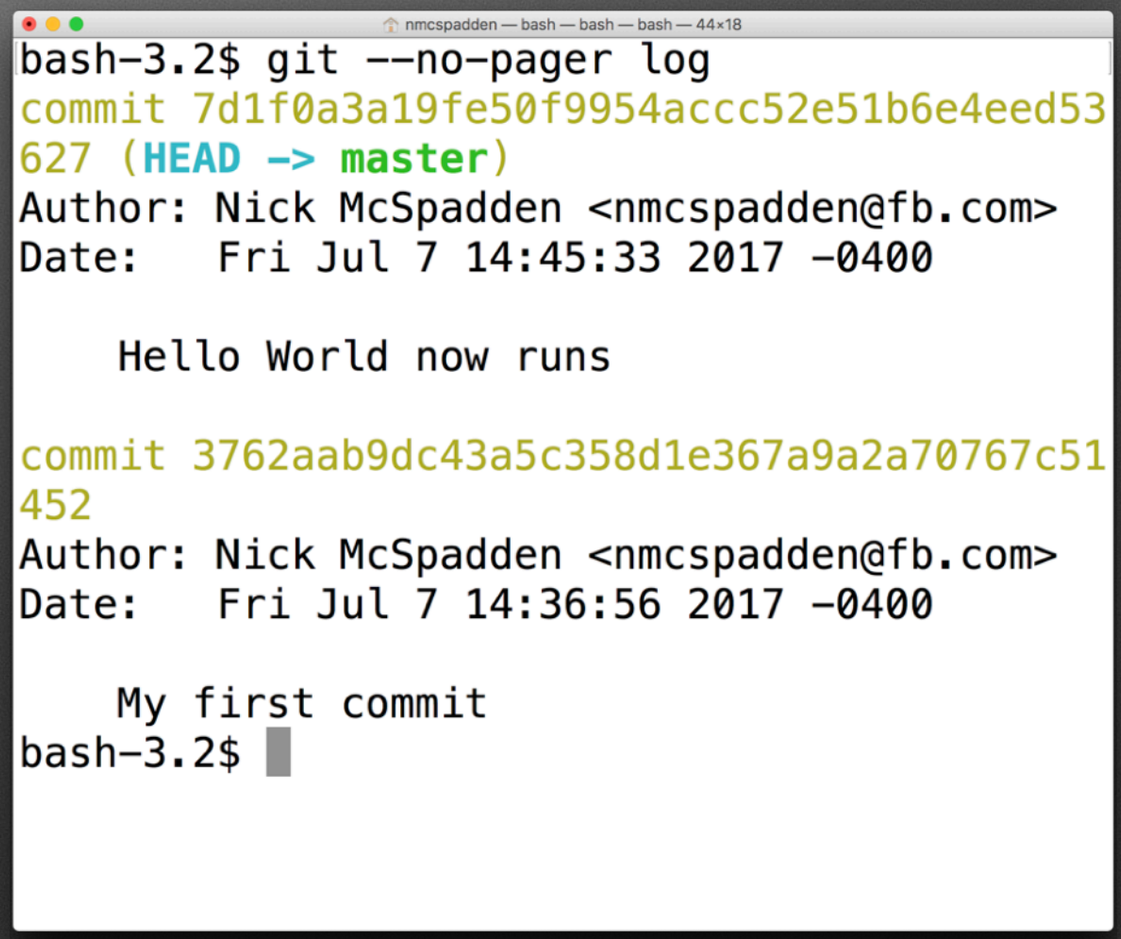
```
$ git commit -am "Hello
World now runs"
```

Let's add some real content

The git log now shows both of our commits:

```
$ git --no-pager log
```

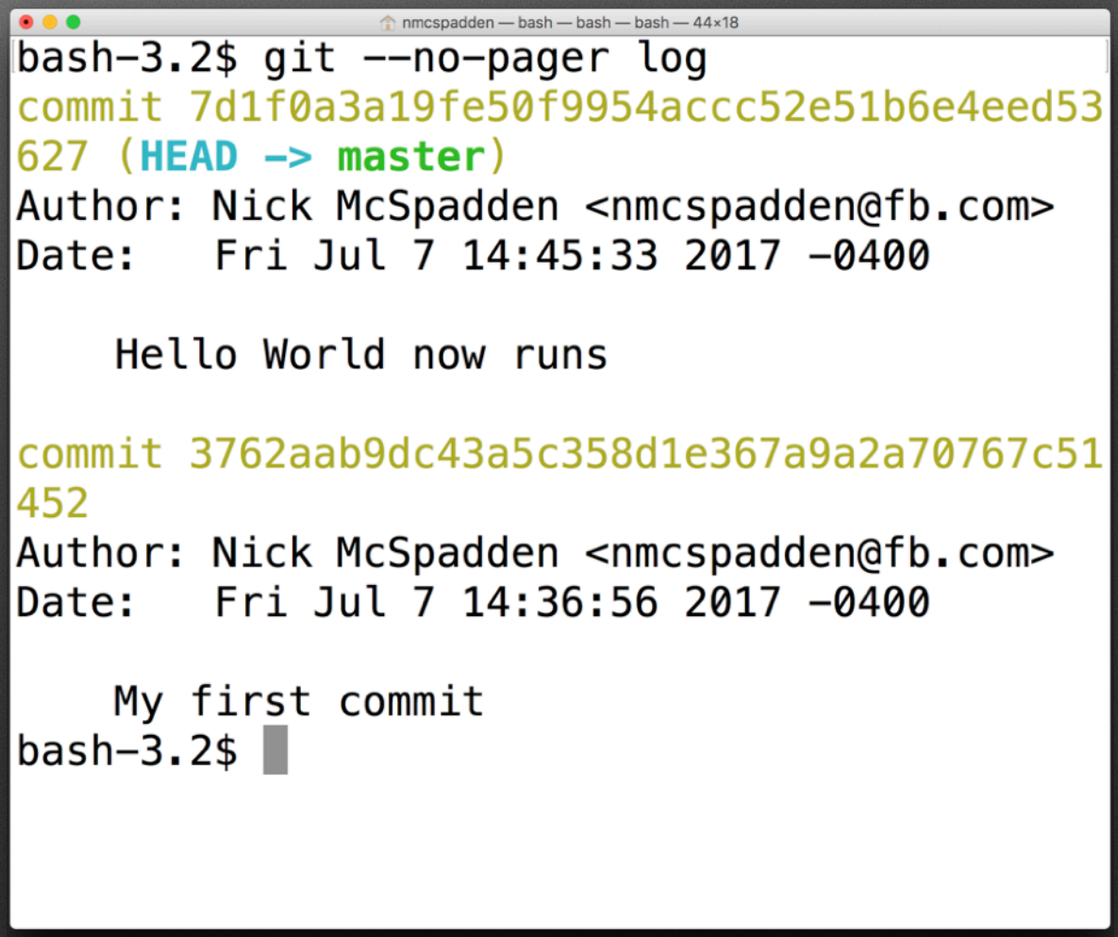
`git log` shows commits in reverse chronological order - newest ones are on top.



Anatomy of `git log`

Commits are referred to by their hash.

The commit hash is the first 7 characters of the SHA2 sum of the changes.

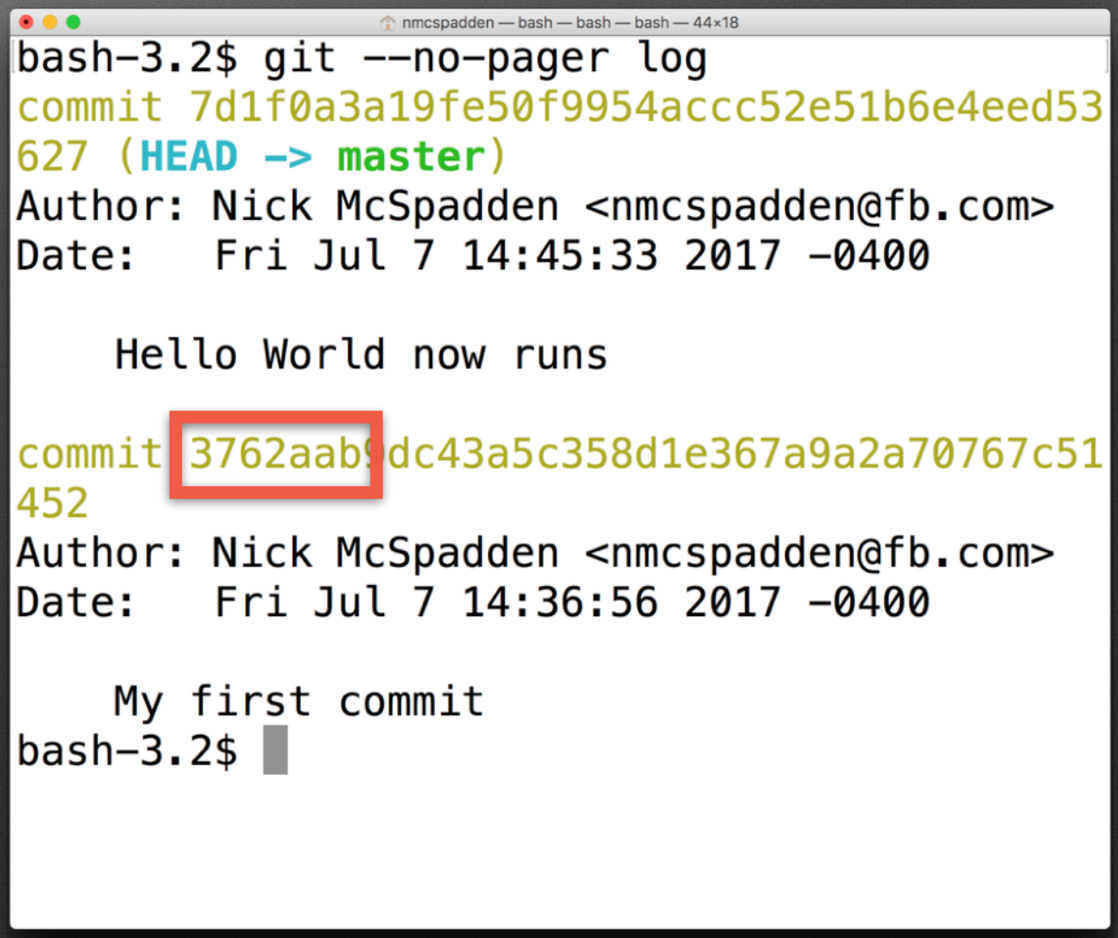


Anatomy of `git log`

Commits are referred to by their hash.

The commit hash is the first 7 characters of the SHA2 sum of the changes.

First commit: 3762aab



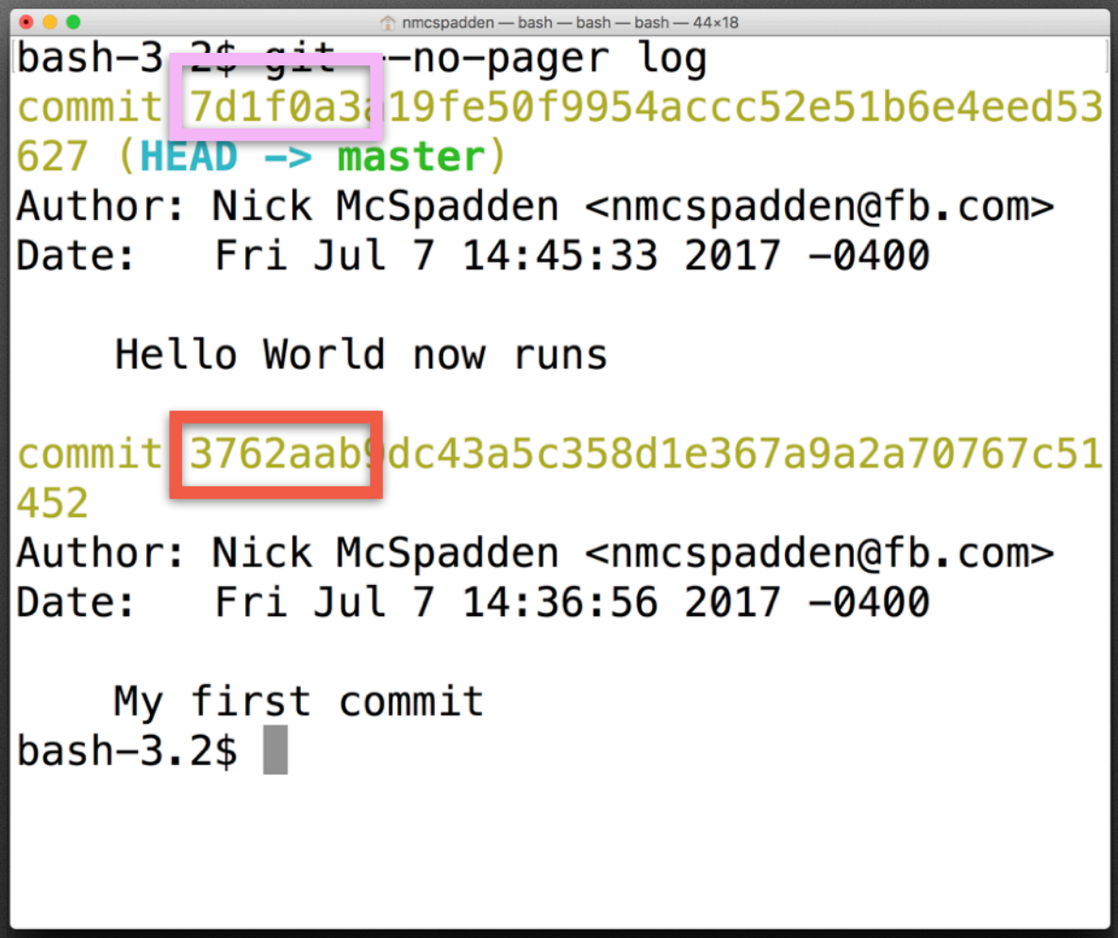
Anatomy of `git log`

Commits are referred to by their hash.

The commit hash is the first 7 characters of the SHA2 sum of the changes.

First commit: 3762aab

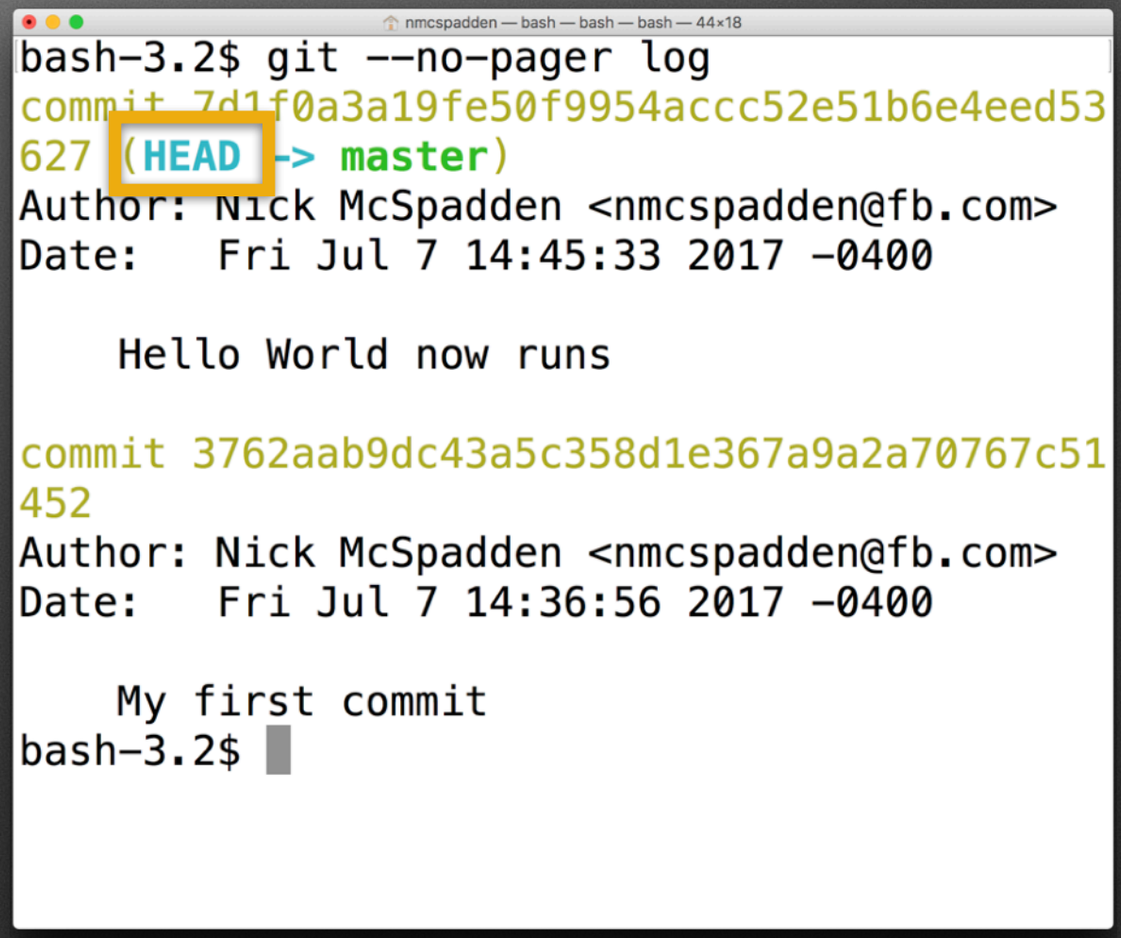
Second commit: 7d1f0a3



Anatomy of `git log`

Where are we right now? The HEAD tells us what commit we are working off of.

Right now, our HEAD is also on master - meaning we are at the tip of the master branch.



We have code in our repo, let's branch out a bit...

Using feature branches

One of `git`'s most powerful techniques is branching.

- Each branch contains completely separate states for files in the repo.
- You can switch between branches at any time.
- `master` is an omnipresent branch that represents the Source of Truth.

Using feature branches

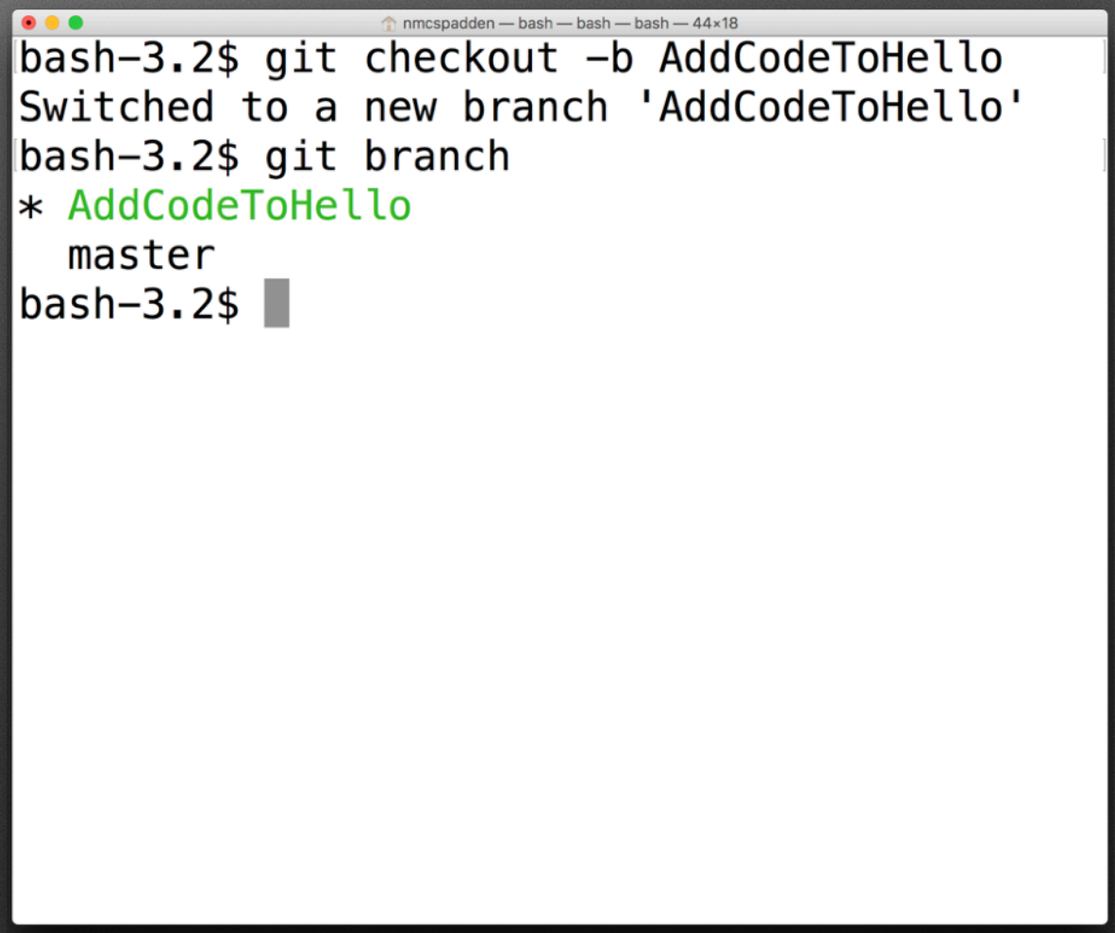
What branches are available?

```
$ git branch
```

Switch to a new branch now:

```
$ git checkout -b
AddCodeToHello
```

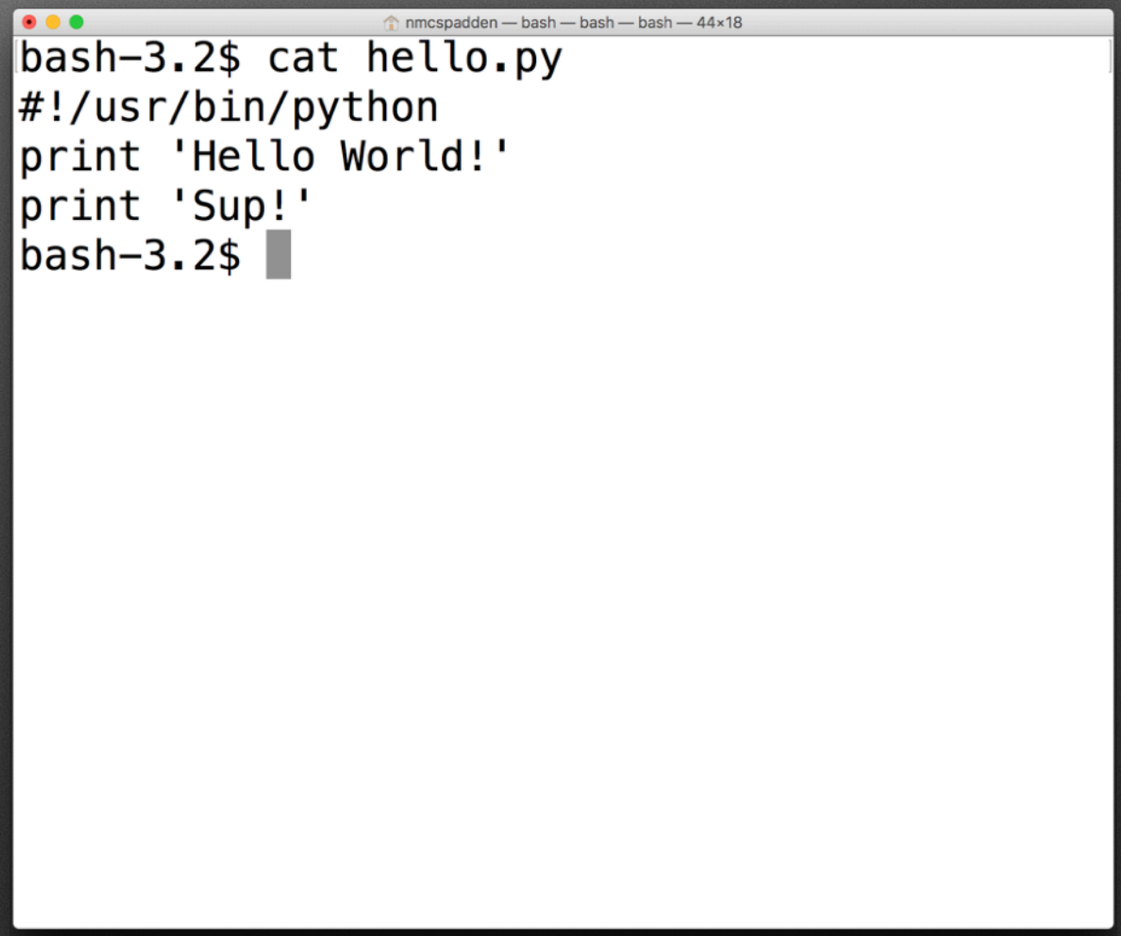
The branch name is for your own convenience and reference. Choose names that make logical sense to you.



Using feature branches

Let's make a quick change to hello.py. Edit the file like a Californian:

```
#!/usr/bin/python

print 'Hello world!'
print 'Sup!'
```

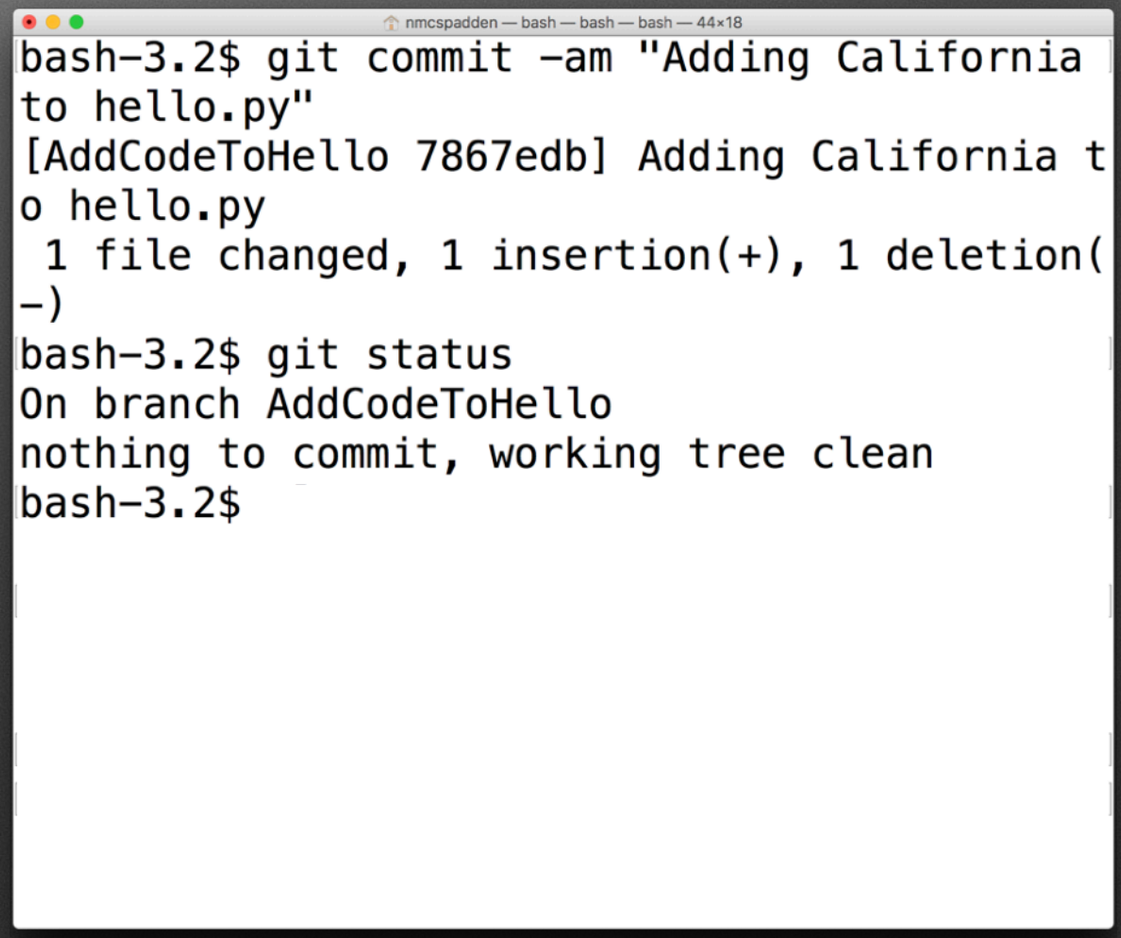


Using feature branches

At this point, we have unstaged changes, so let's go ahead and make a commit in this branch:

```
$ git commit -am "Adding California to hello.py"
```

After this, we'll be in a clean state on our branch.



Using feature branches

Want a truly visual indication of how branches work?

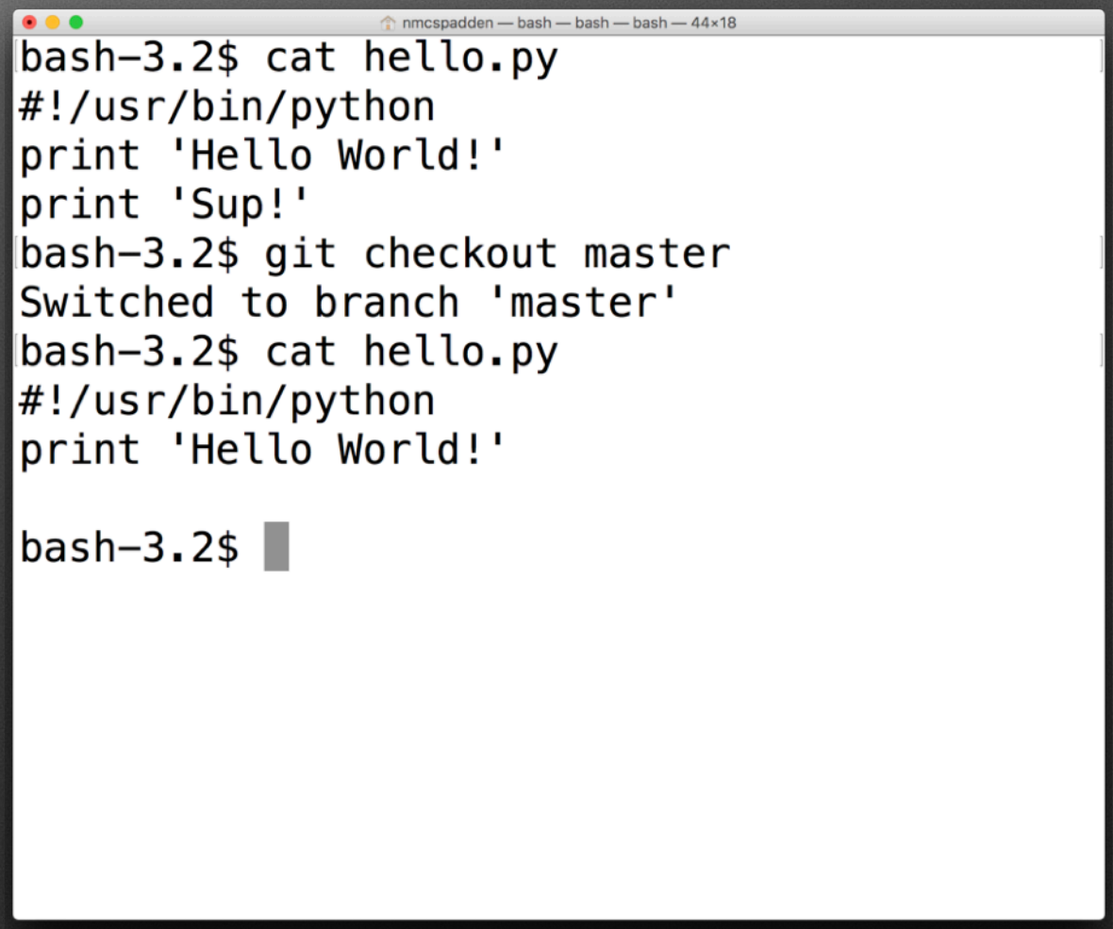
```
$ cat hello.py
```

Now switch back to master:

```
$ git checkout master
```

```
$ cat hello.py
```

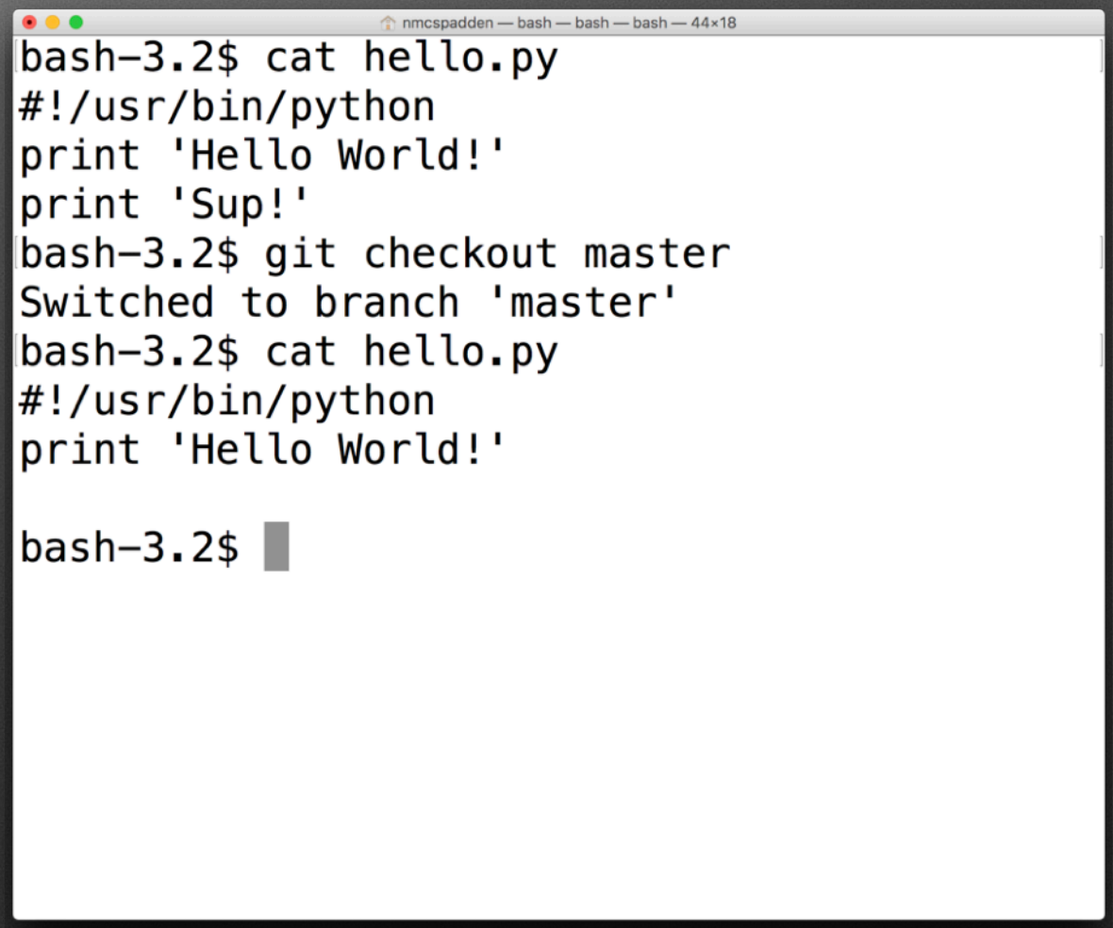
If you have an editor open, you might see the file change immediately.



Using feature branches

Checking out branches will cause `git` to update all the files to its snapshot according to its history.

It's like switching to a Time Machine snapshot for the entire project tree at once.



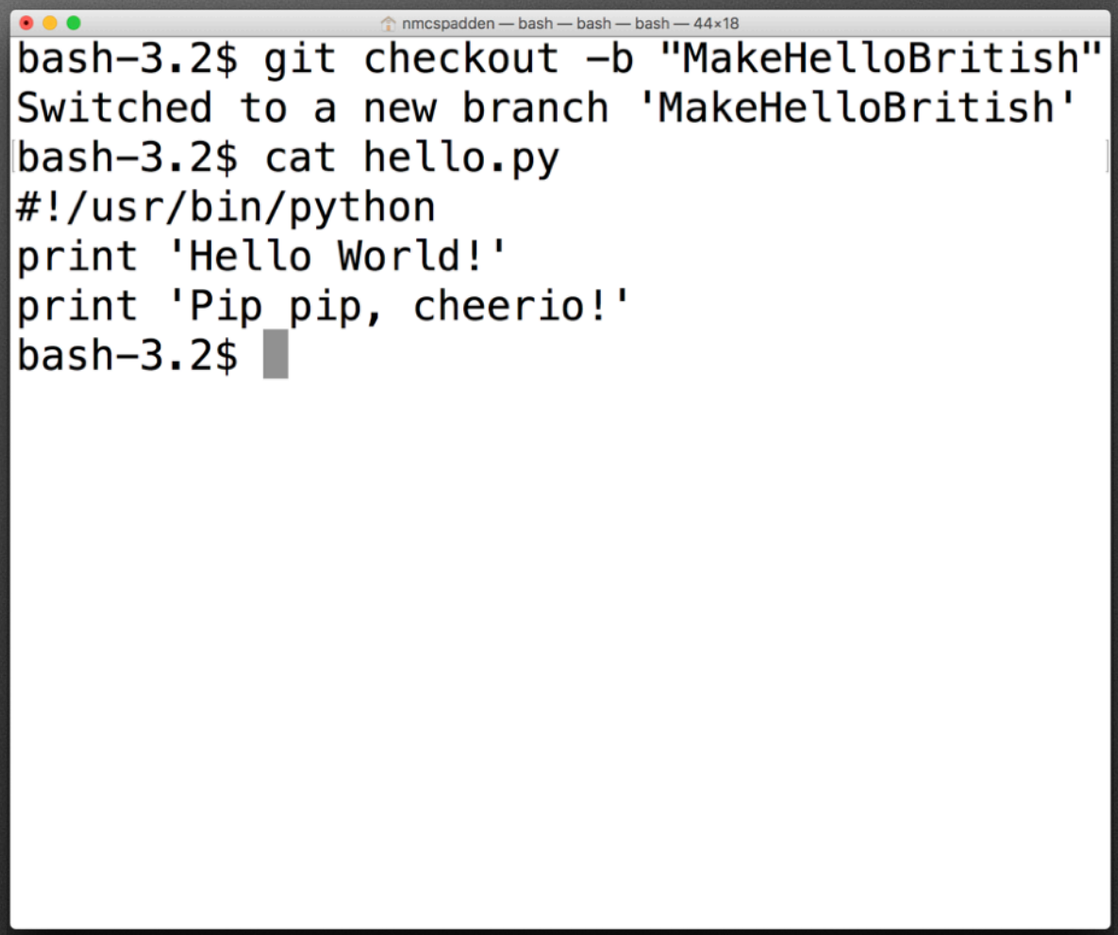
Using feature branches

Let's go a step further, and make a new branch with *different* changes.

```
$ git checkout -b
"MakeHelloBritish"
```

Edit hello.py to make it more British:

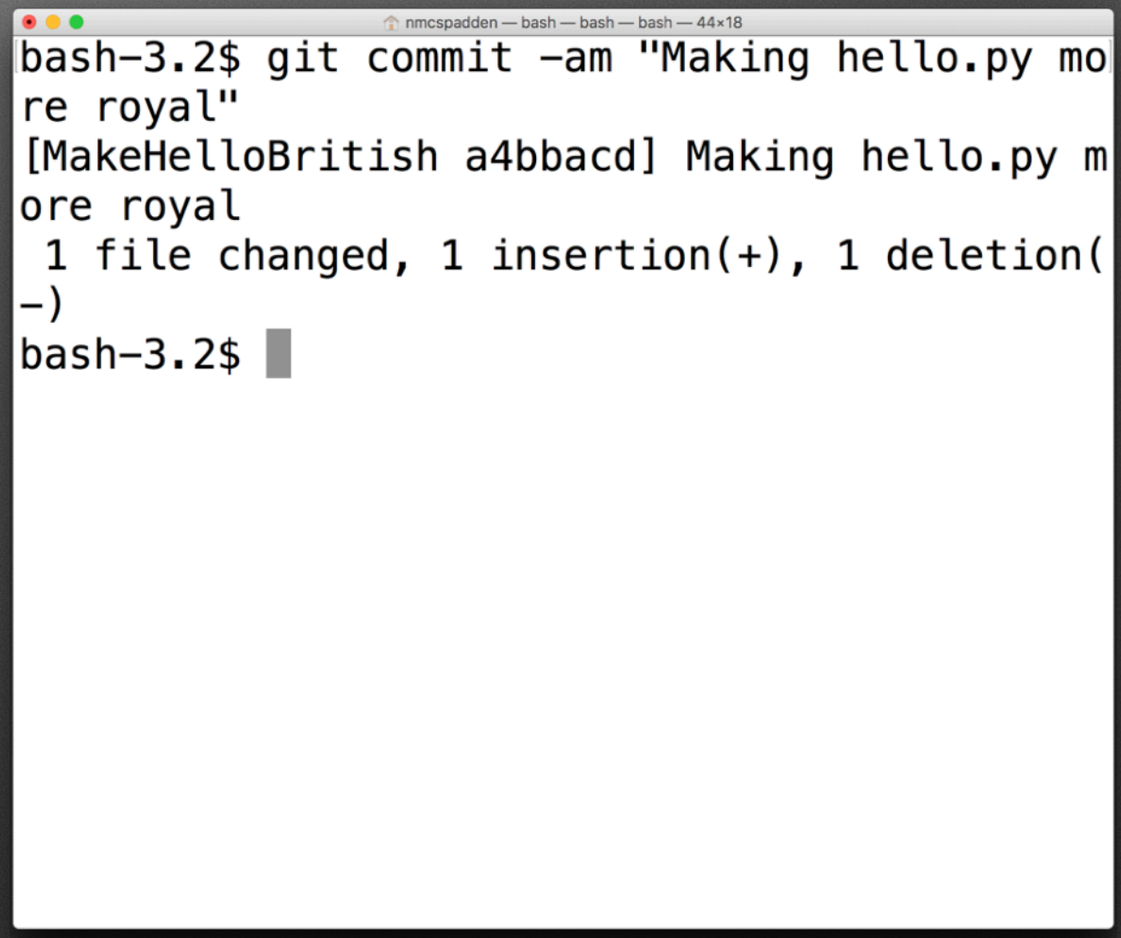
```
#!/usr/bin/python

print 'Hello World!'
print 'Pip pip, cheerio!'
```



Using feature branches

Again, we need to commit this change:

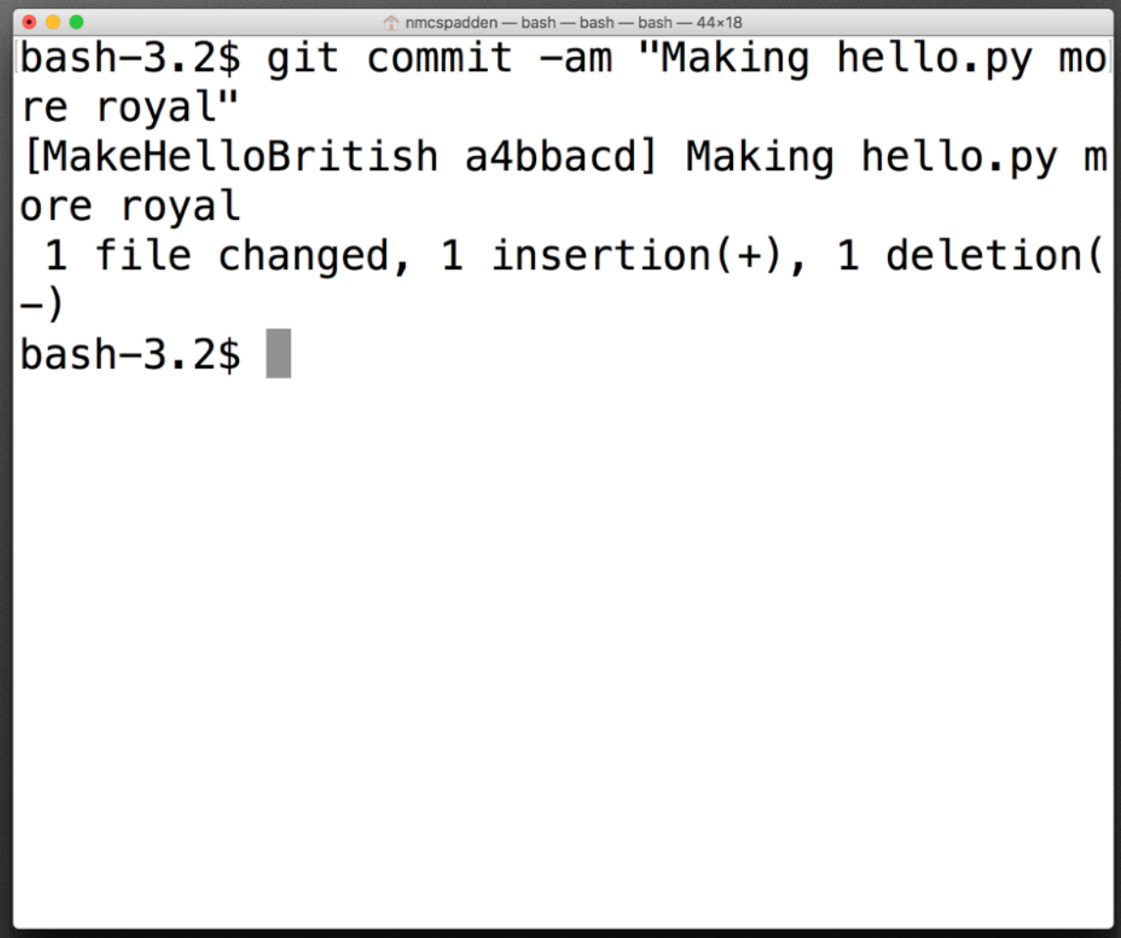
```
$ git commit -am "Making hello.py more royal"
```

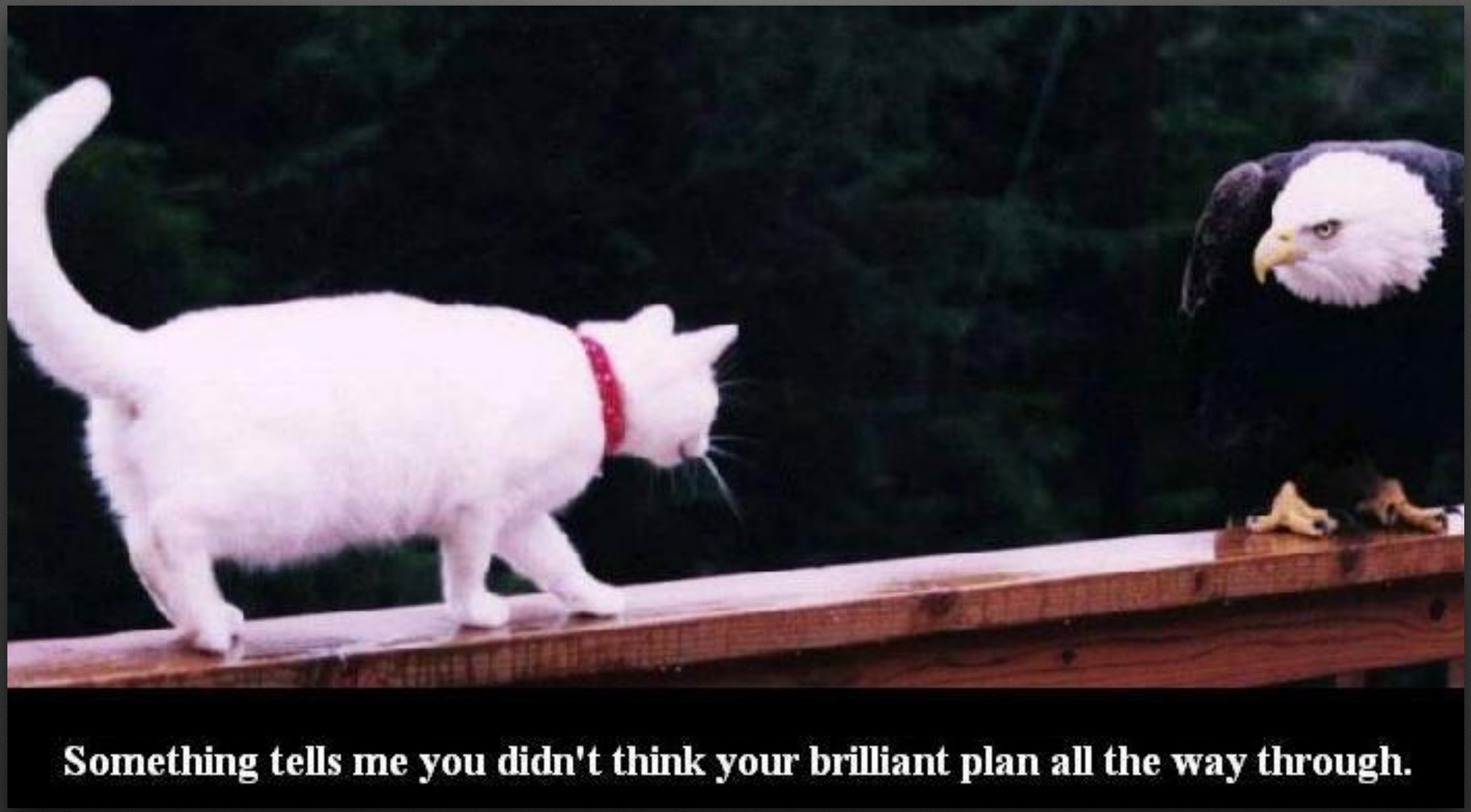


Using feature branches

Again, we need to commit this change:

```
$ git commit -am "Making hello.py more royal"
```





So now we have two different branches, both with changes to the same file on the same line...

Merging back to master

We have two branches. We want to `merge` our changes into master.

Because we're masochists, we're going to `merge` both branches into master, one at a time, even though we know this is not a good plan.

But it makes for a hilarious demo!

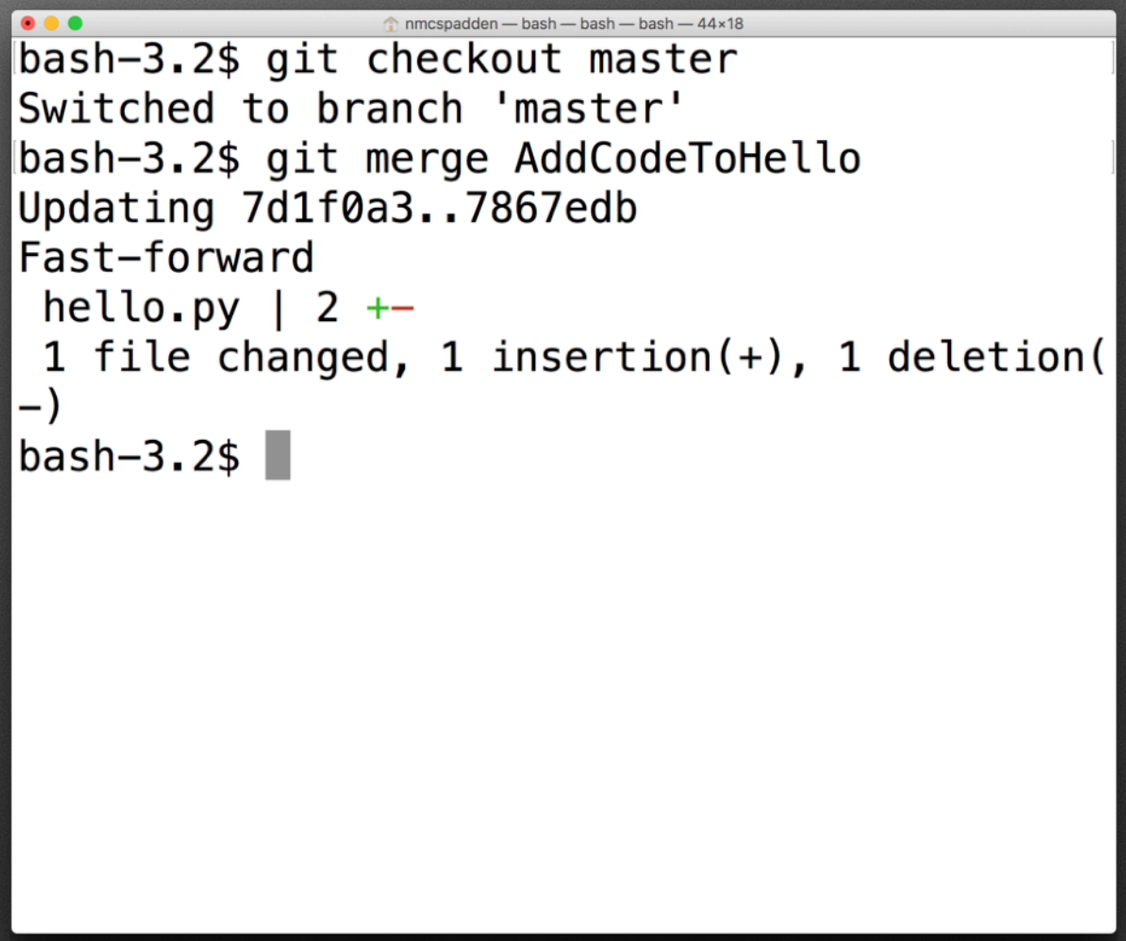
master

A visual metaphor for `git merge`s

Merging back to master

Merge the California branch back into master. Switch back to master, and then use `git merge`:

```
$ git checkout master

$ git merge
AddCodeToHello
```



Merging back to master

If you look in the `git log`, you can see our feature branch was incorporated into master:

```
$ git --no-pager log
```

```
nmcspadden — bash — bash — bash — 44×18
bash-3.2$ git --no-pager log
commit 7867edb5da26ea2b185143b8f57368b08ad90bea (HEAD -> master, AddCodeToHello)
Author: Nick McSpadden <nmcspadden@fb.com>
Date:   Sat Jul 8 18:59:54 2017 -0400

    Adding California to hello.py

commit 7d1f0a3a19fe50f9954accc52e51b6e4eed53627
Author: Nick McSpadden <nmcspadden@fb.com>
Date:   Fri Jul 7 14:45:33 2017 -0400

    Hello World now runs

commit 3762aab9dc43a5c358d1e367a9a2a70767c51452
Author: Nick McSpadden <nmcspadden@fb.com>
```

Merging back to master

Let's clean up after ourselves, since we won't need that branch anymore:

```
$ git branch -d
AddCodeToHello
```

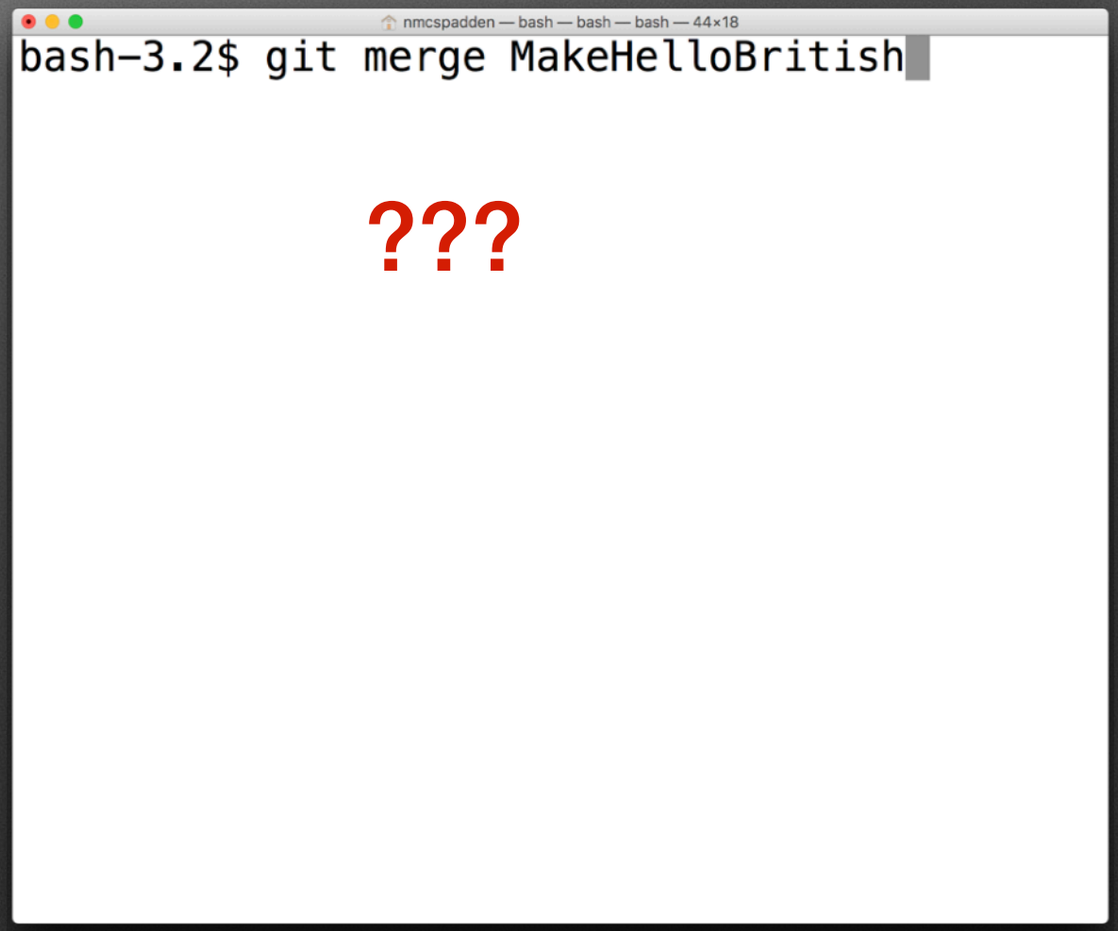
Merging back to master

So that worked.

Now let's do it again with our British branch!

```
$ git merge
MakeHelloBritish
```

This probably ain't gonna work so well...



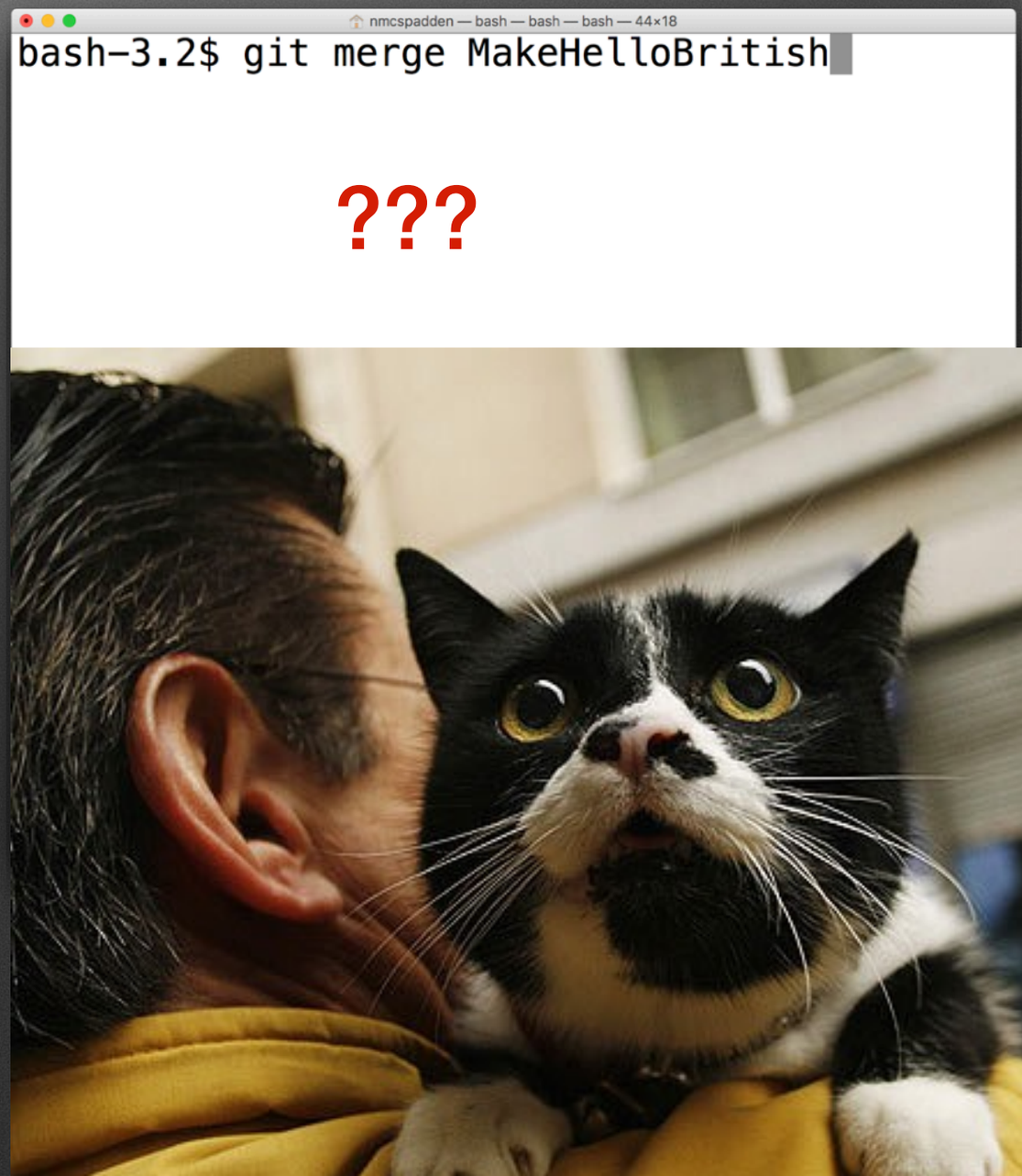
Merging back to master

So that worked.

Now let's do it again with our British branch!

```
$ git merge
MakeHelloBritish
```

This probably ain't gonna work so well...



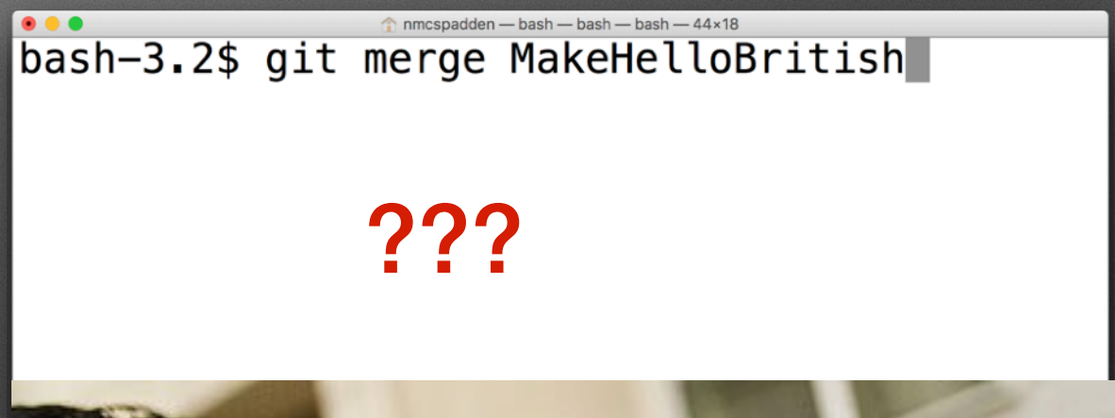
Merging back to mas

So that worked.

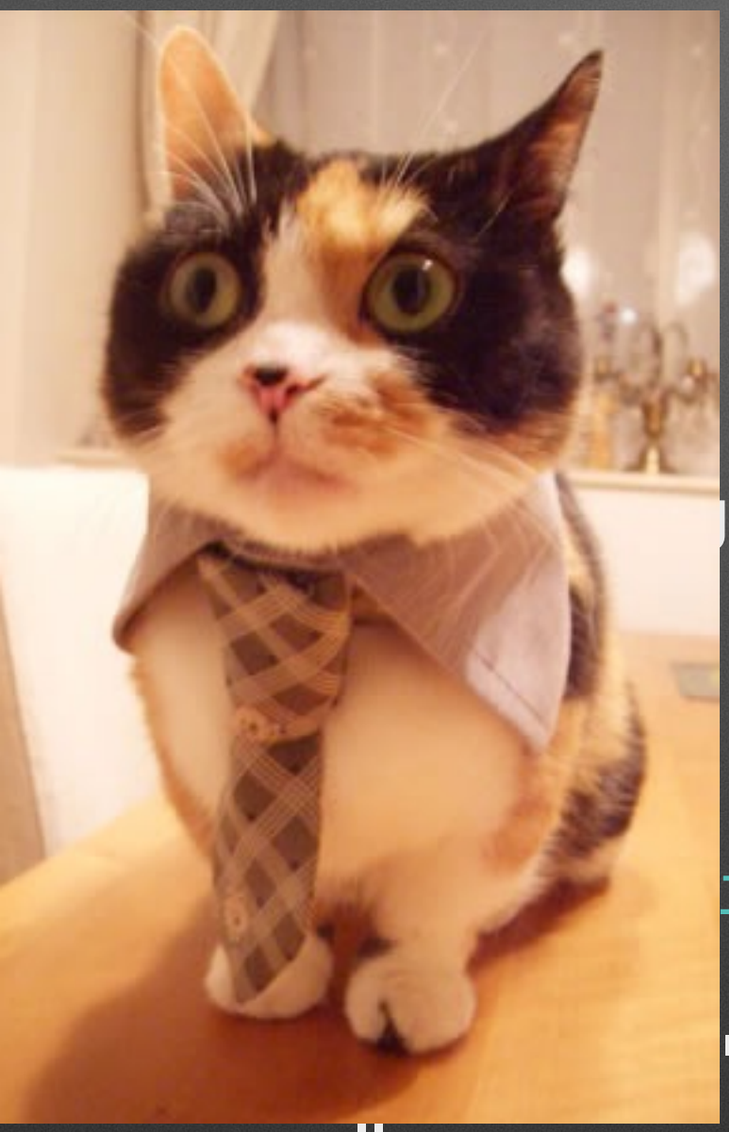
Now let's do it again with our British branch!

```
$ git merge
MakeHelloBritish
```

This probably ain't gonna work so well...



Merging back to mas



ain with our

ish

't gonna work

so well...

nmcspadden — bash — bash — bash — 44×18

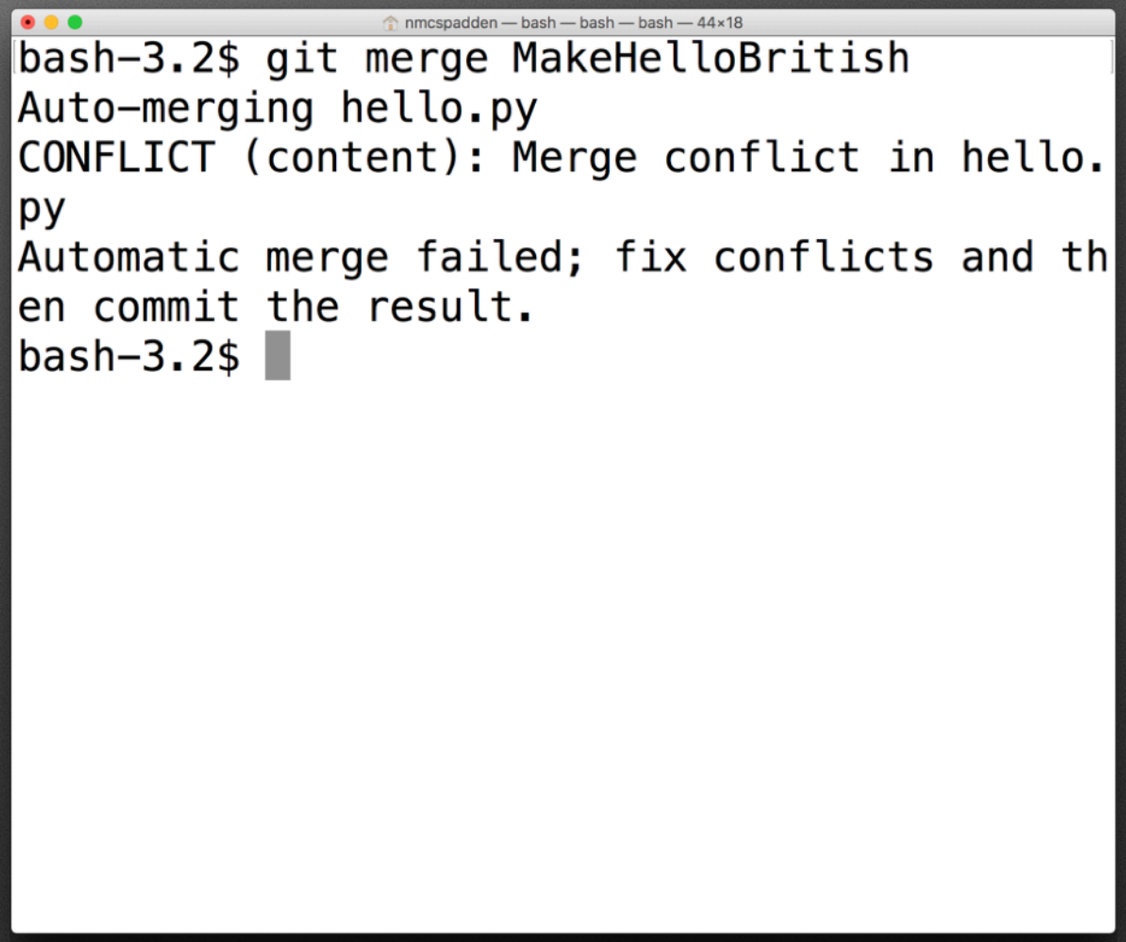
```
bash-3.2$ git merge MakeHelloBritish
```

???

Merging back to master

```
$ git merge
MakeHelloBritish
```

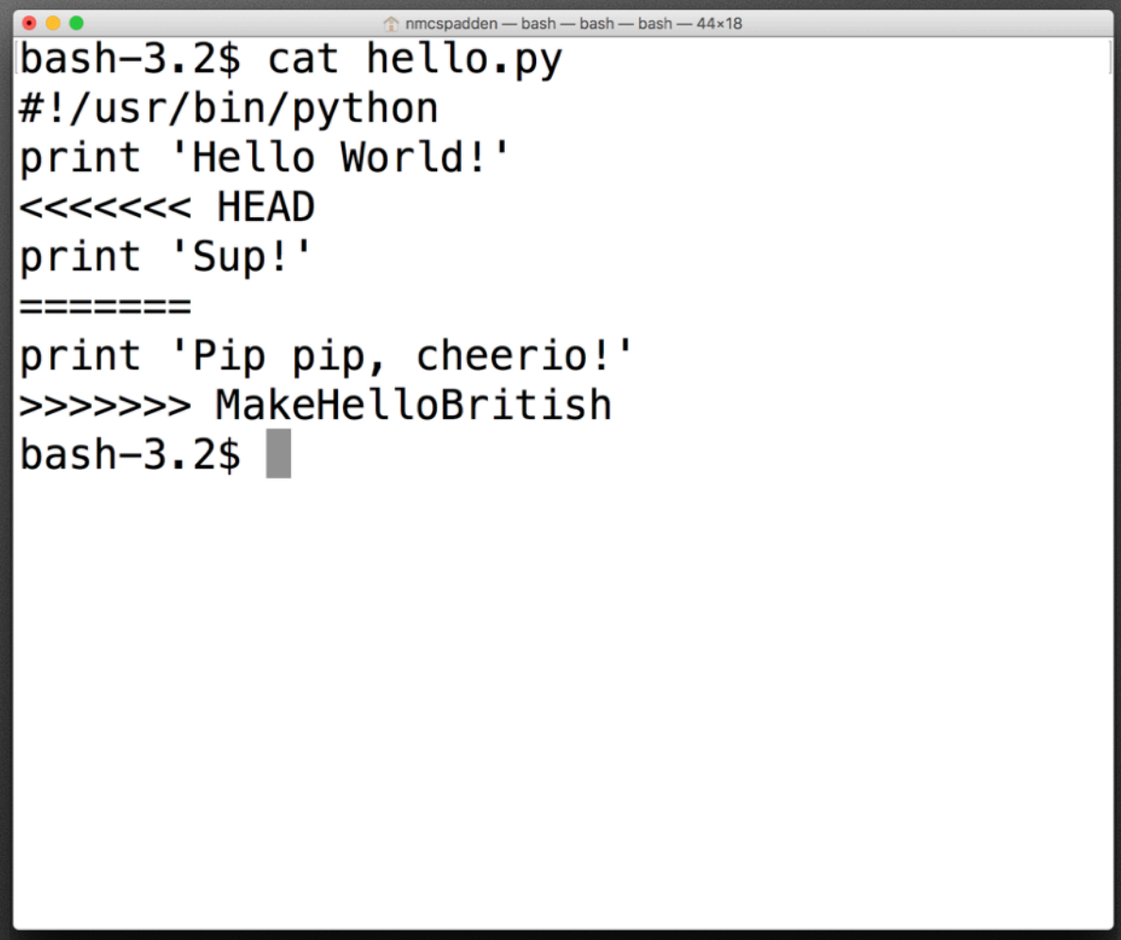
Merge conflict!



Merge conflicts

Conflict resolution

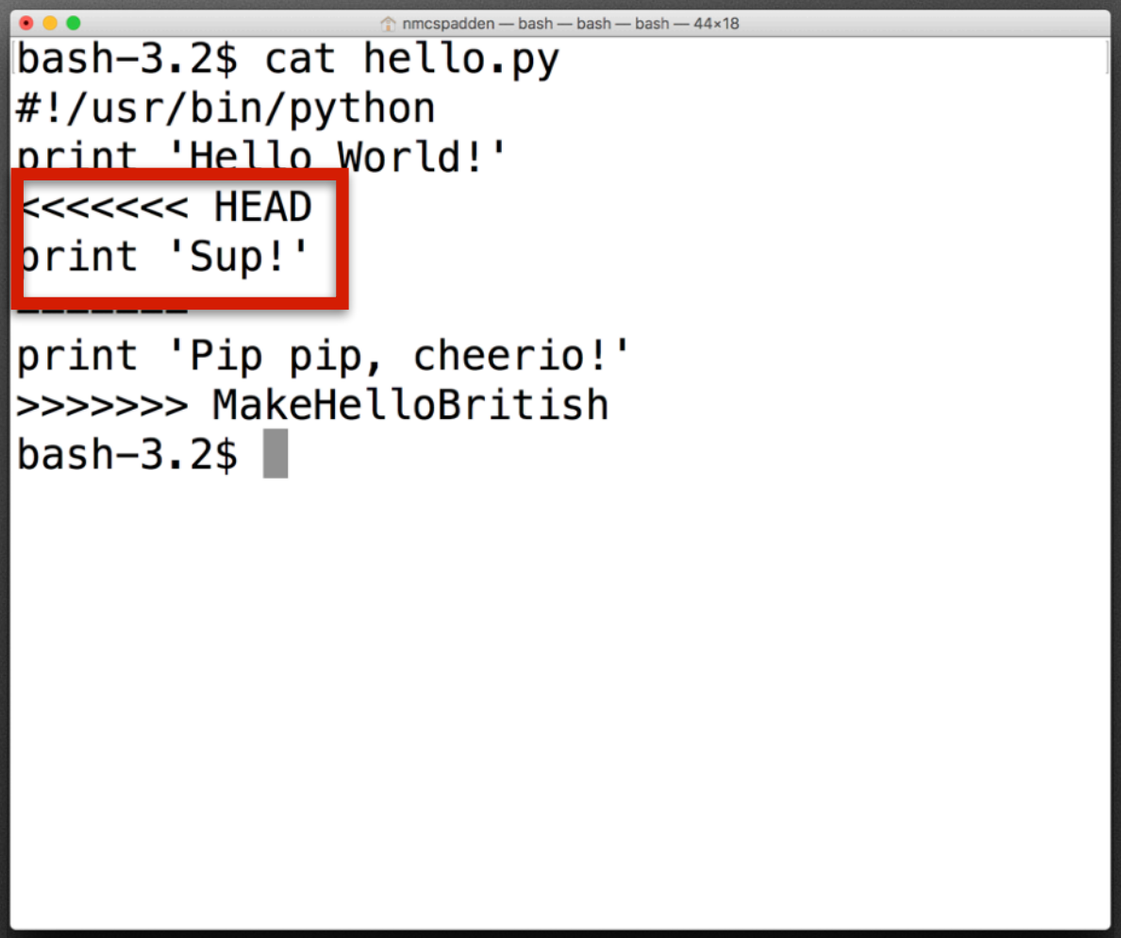
If you look at hello.py, you'll see that `git` has inserted "conflict markers" to indicate which lines are currently in conflict.



Conflict resolution

Conflict markers show you the conflict between your merge attempt.

"HEAD" tells you what's currently in the file at that line.

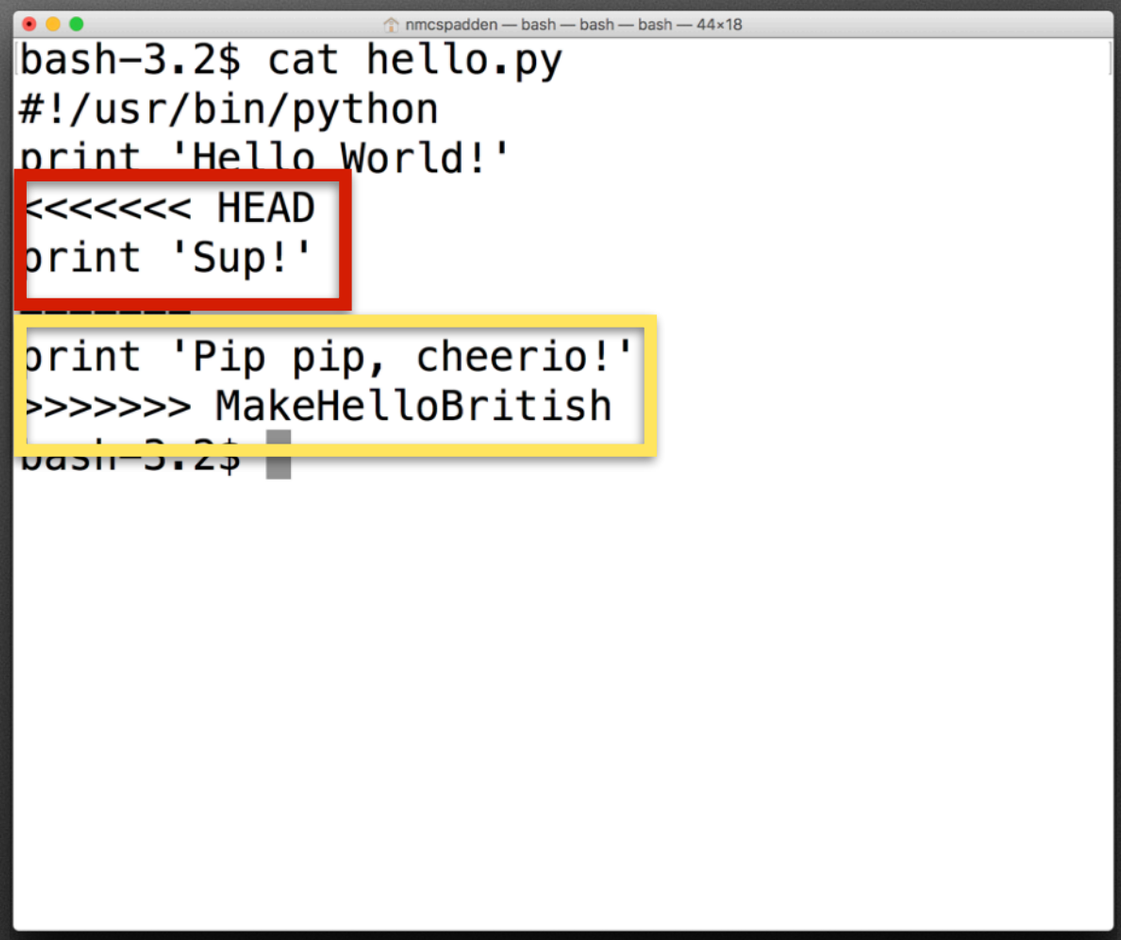


Conflict resolution

Conflict markers show you the conflict between your merge attempt.

"<<<<HEAD" tells you what's currently in the file at that line (i.e. what you're trying to merge *into*).

">>>>branch" tells you what your change is (i.e. what you're trying to merge *from*).

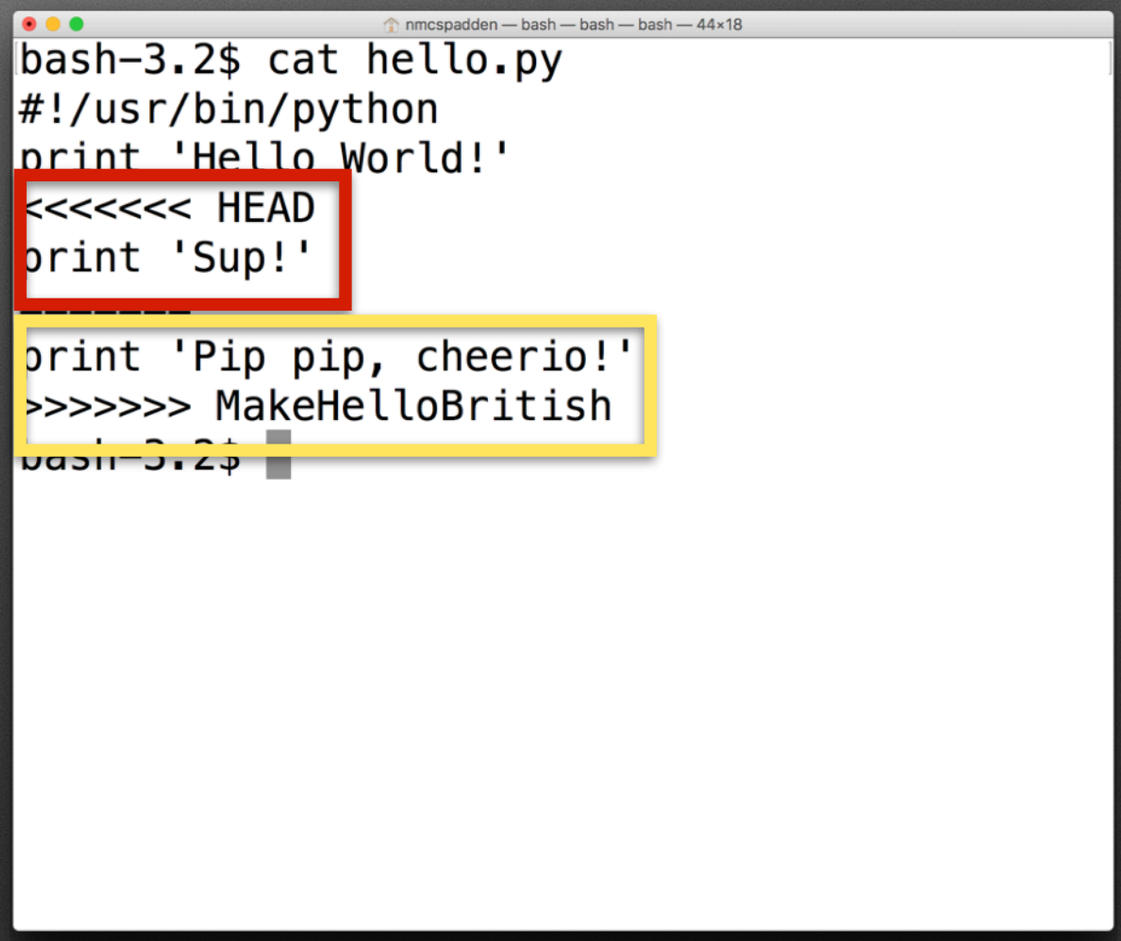


Conflict resolution

`git` will never, ever make decisions on your behalf.

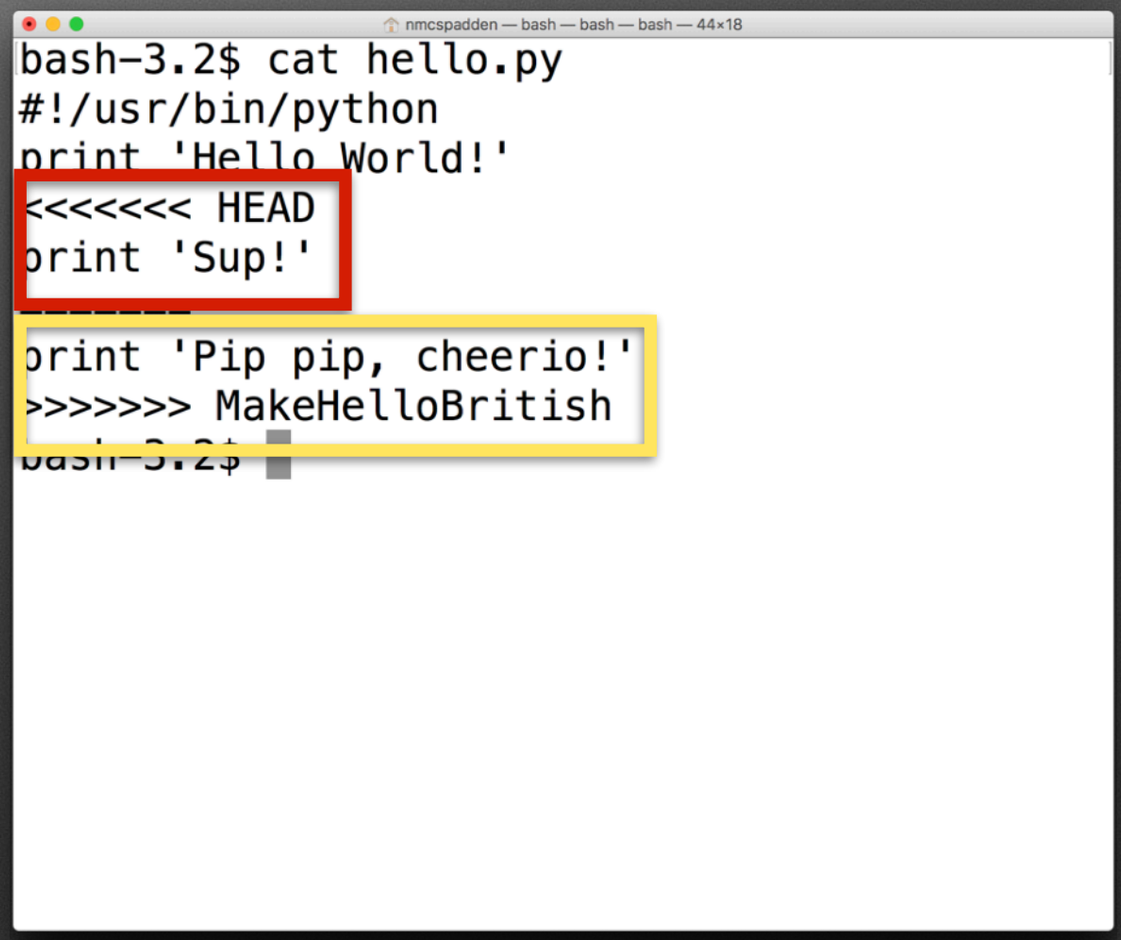
If a decision has to be made, you must resolve it before continuing.

So you must now decide which version you want to keep - Californian or British?



Conflict resolution

Resolving a conflict is easy*: delete all the conflicted parts and keep only the chunks you want.



* This is a complete and utter lie.

Conflict resolution

Step 1

```
1  #!/usr/bin/python
2  print 'Hello World!'
3  <<<<<<< HEAD
4  print 'Sup!'
5  =======
6  print 'Pip pip, cheerio!'
7  >>>>>>> MakeHelloBritish
8
```

I'm going to keep the Californian version, so I'm just going to delete the extra stuff.

Step 2

```
1  #!/usr/bin/python
2  print 'Hello World!'
3  print 'Sup!'
4
```

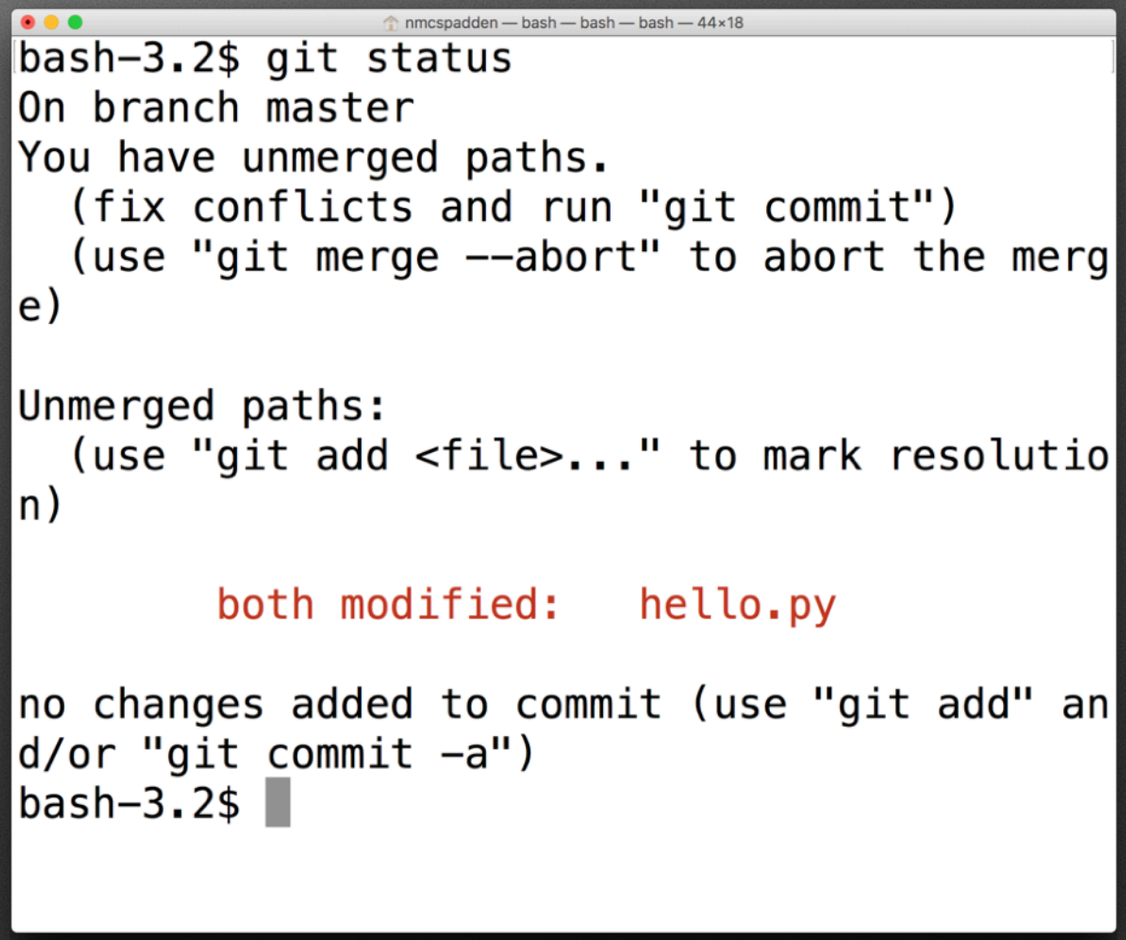
Conflict resolution

Once you've saved your changes, you see can that `git status` tells you what you can do:

```
$ git status
```

It tells you its suggestion right at the top:

```
fix conflicts and run "git commit"
```



Conflict resolution

```
$ git commit -a
```

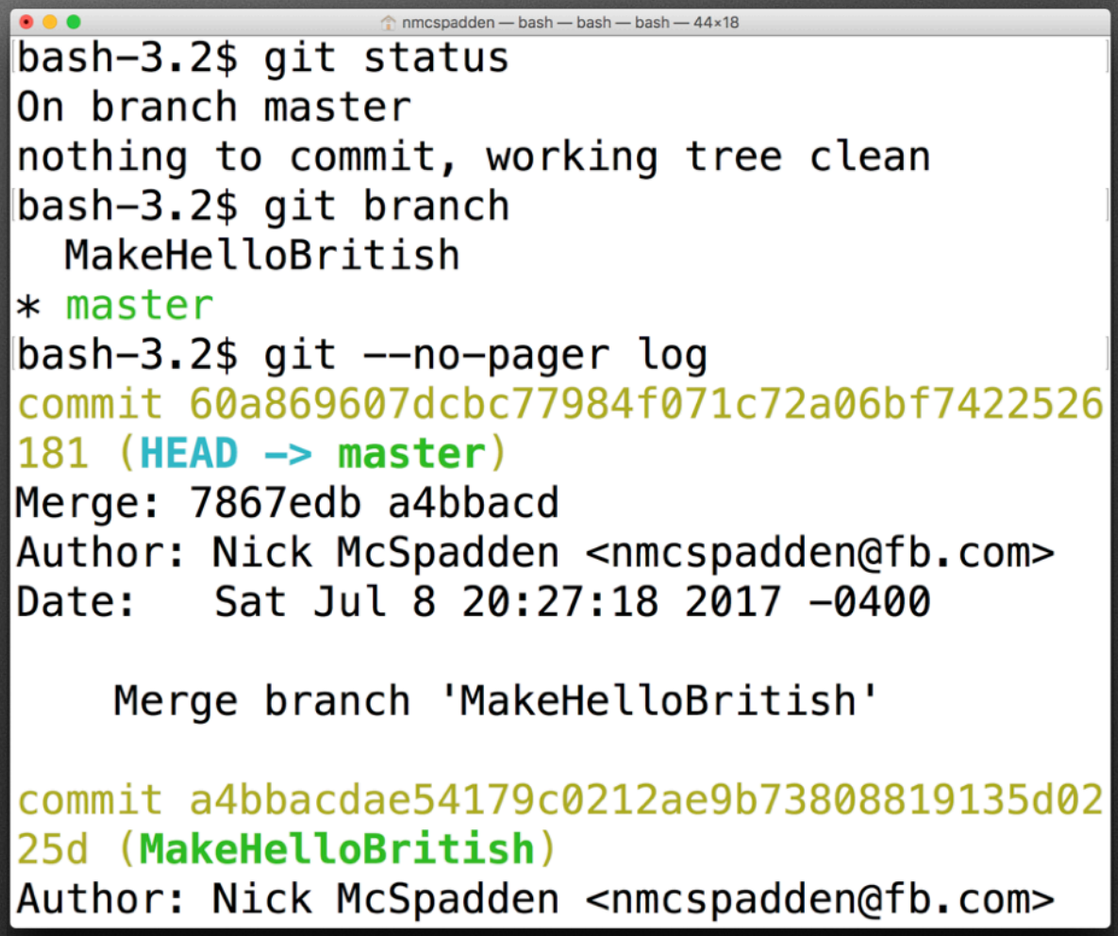
Now you'll have to edit your commit message, because it's created a merge commit for you.

Usually, you can just roll with whatever it gives you without changes.

Conflict resolution

We're done! The log shows we merged successfully (even if we didn't actually do anything):

```
$ git status

$ git branch

$ git --no-pager log
```



But what about... GitHub?

Now you can f#$& up *even faster*.

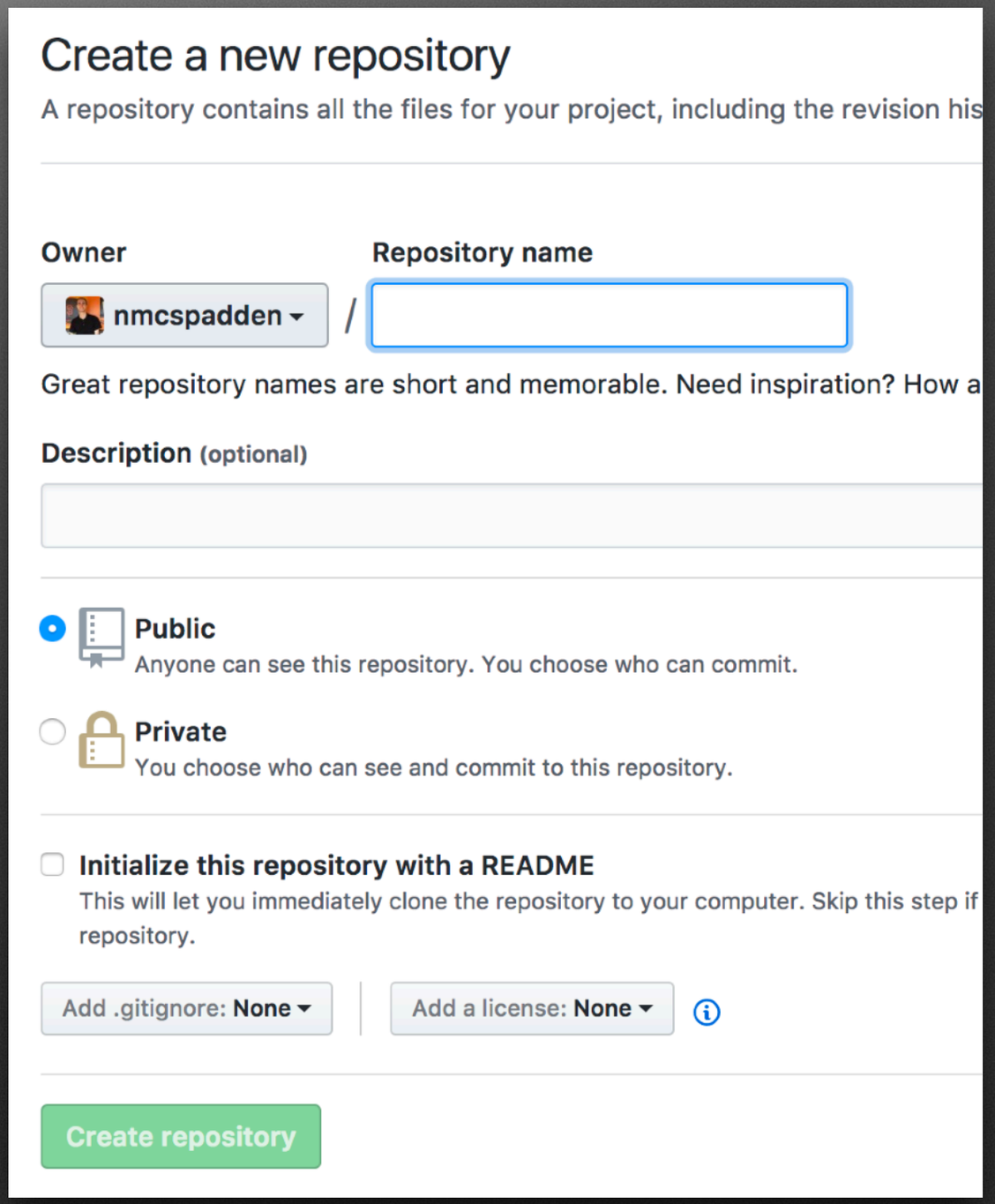
What is GitHub?

- Cloud-based git repositories!
- Many popular open source projects in the Mac community are hosted here - Munki, AutoDMG, Imagr
- Offers a lot of conveniences in a web UI for common `git` operations

Moving to GitHub

We already have an existing git repo, with something in it. Let's push it to GitHub!

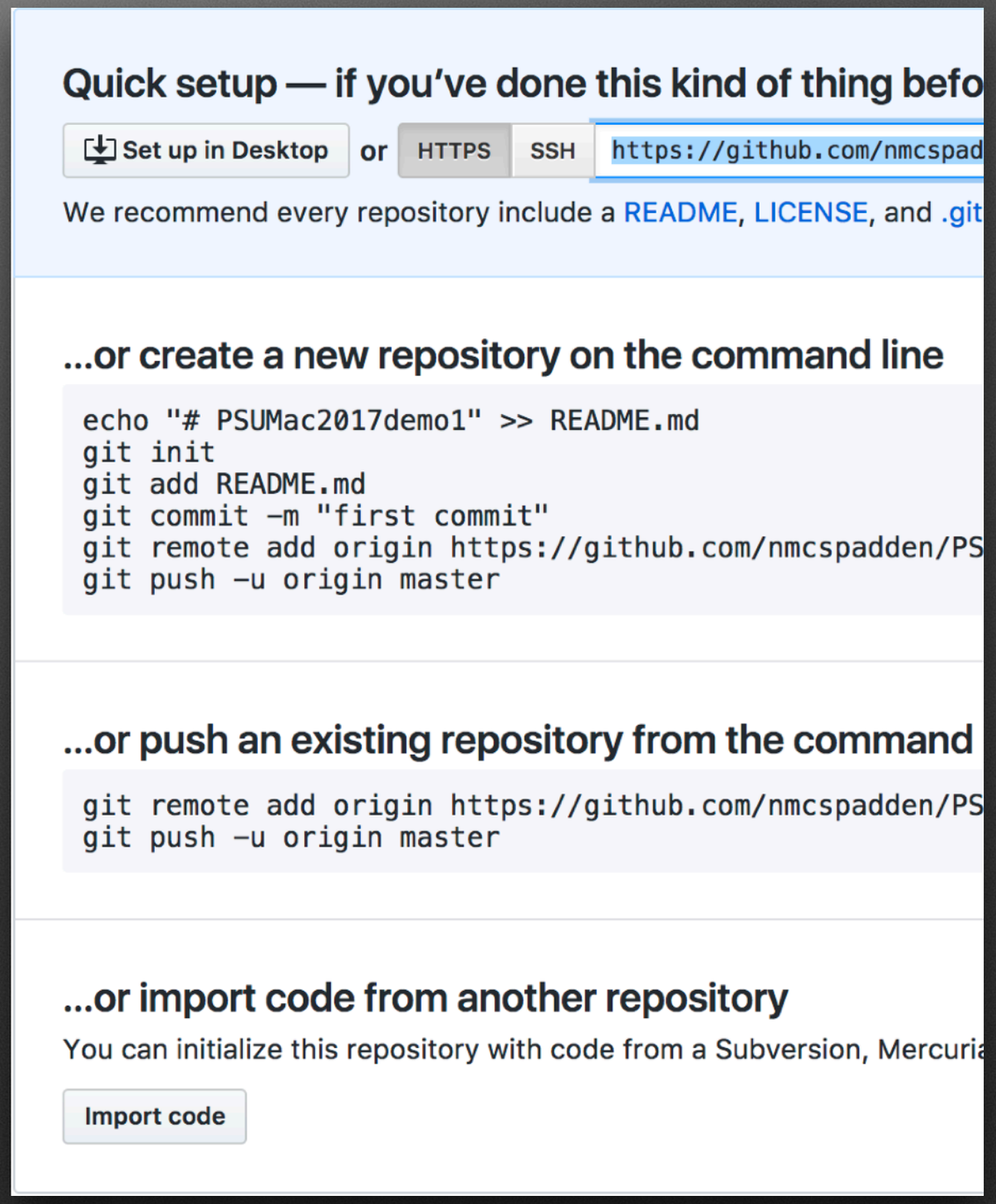
https://github.com/new



Moving to GitHub

https://help.github.com/articles/adding-an-existing-project-to-github-using-the-command-line/

```
$ git remote add
https://github.com/
nmcspadden/
PSUMac2017demo1.git
```



Moving to GitHub

https://help.github.com/articles/adding-an-existing-project-to-github-using-the-command-line/

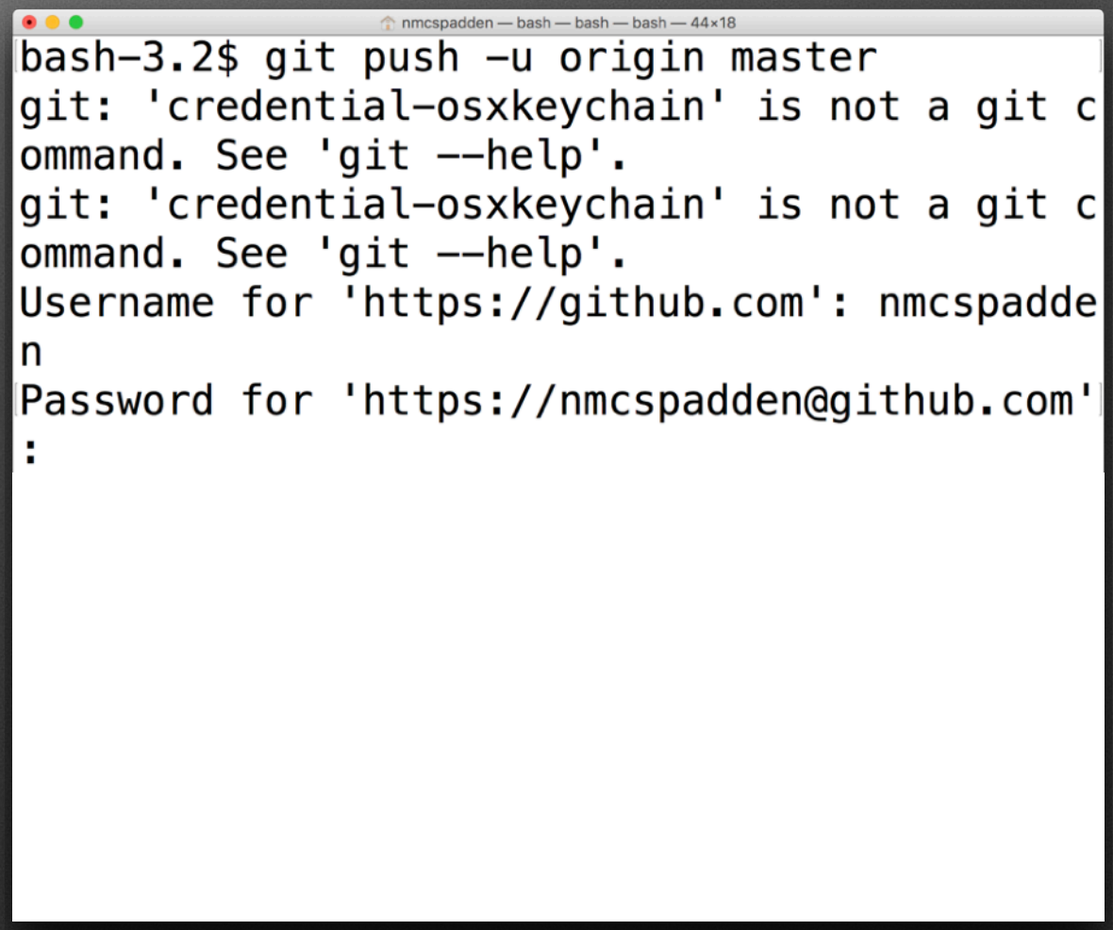
```
$ git remote add origin https://github.com/nmcspadden/PSUMac2017demo1.git
```

"origin" = the remote server on GitHub (i.e. the Source of Truth)

Moving to GitHub

```
$ git push -u origin
master
```

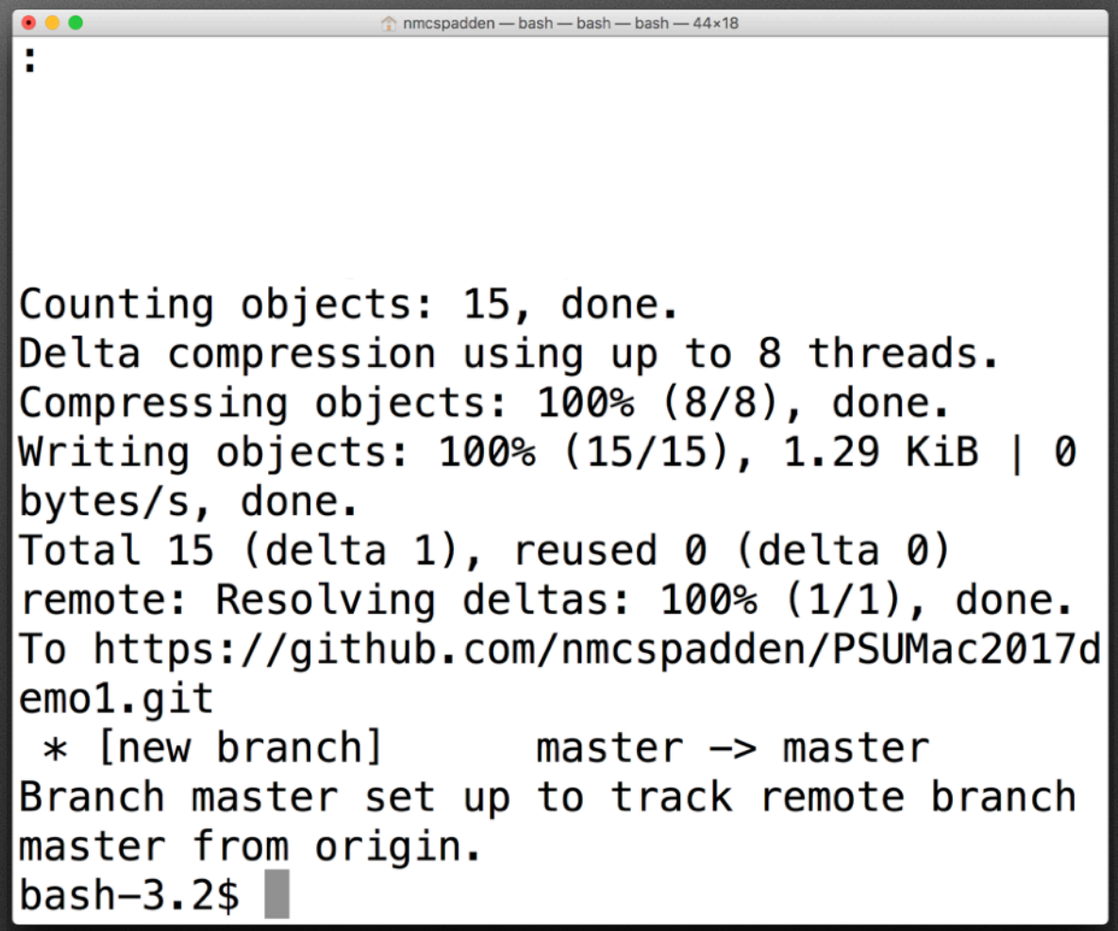
If this is the first time, you'll need to authenticate GitHub.

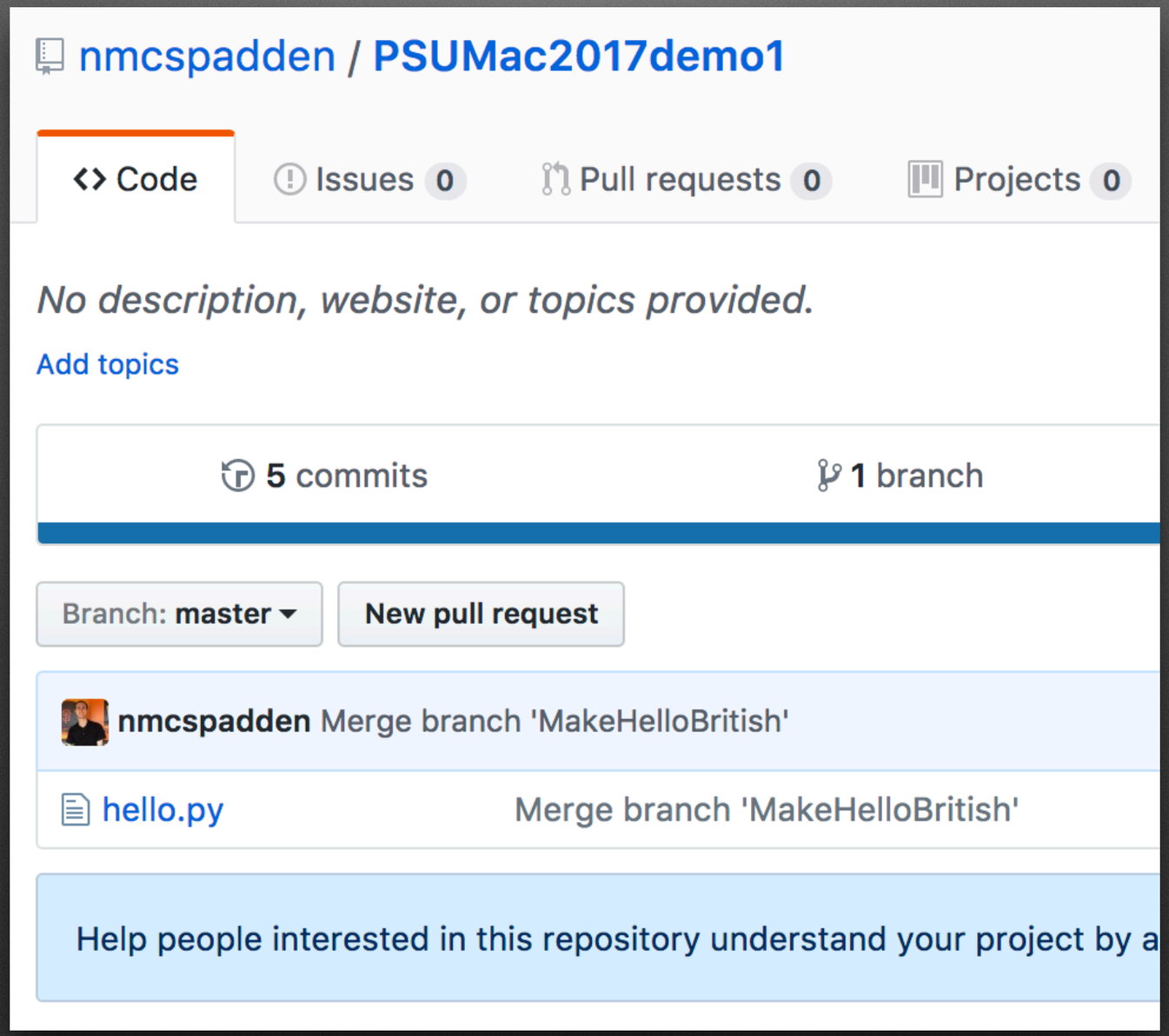


Moving to GitHub

```
$ git push -u origin
master
```

If this is the first time, you'll need to authenticate GitHub.




nmcspadden / PSUMac2017demo1
Code
Issues 0
Pull requests 0
Projects 0
No description, website, or topics provided.
Add topics
5 commits
1 branch
Branch: master
New pull request
nmcspadden Merge branch 'MakeHelloBritish'
hello.py
Merge branch 'MakeHelloBritish'
Help people interested in this repository understand your project by a

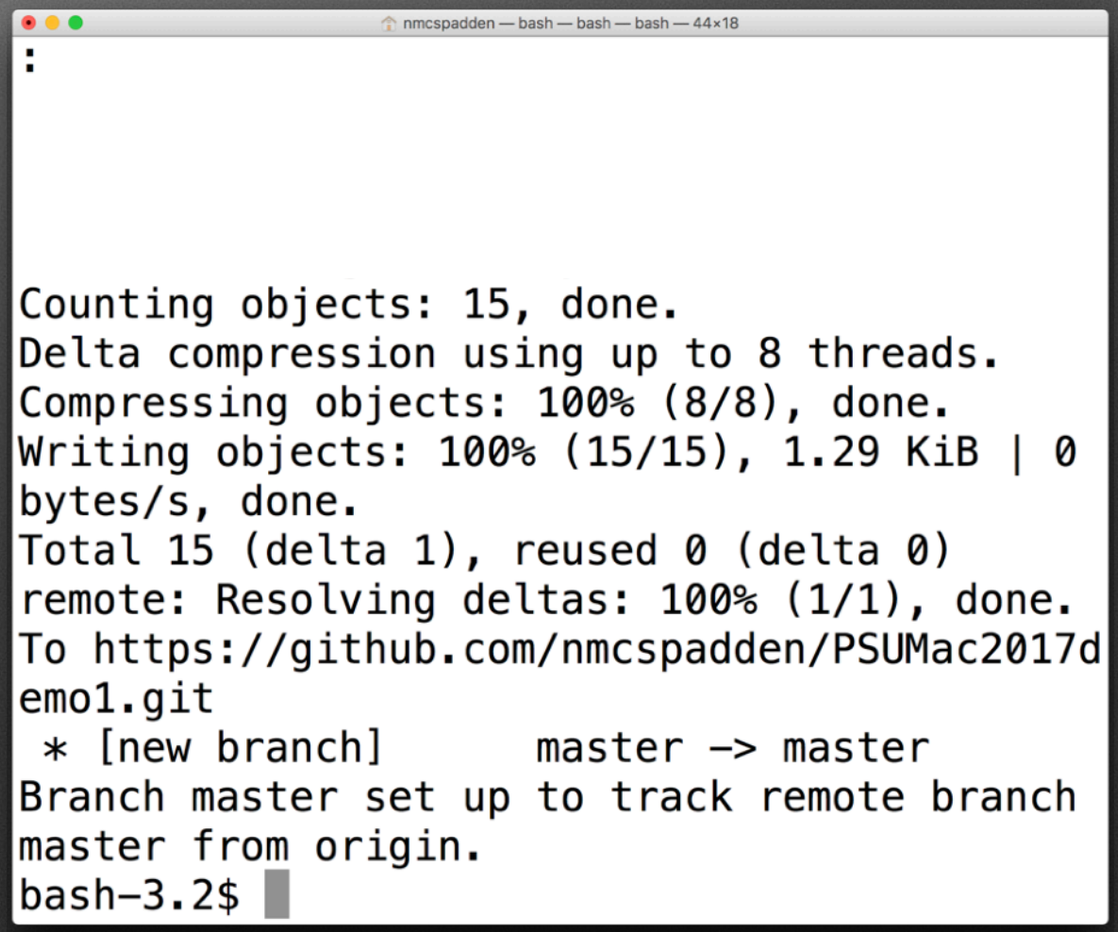
Shiny!

Moving to GitHub

The main differences between local-only git and GitHub:

- Use pull requests!
  When you make a feature branch, you submit a PR to the original repo to pull your branch's changes into their master.

- Rewriting history is so, so, so much worse when other people are involved.



The basic tenet of `git`

"To err is human, but to really f$@# up, you need git" - some wise fellow on the interwebs

The basic tenets of `git`

- Don't work in master. Use feature branches to develop features.
- It is never appropriate to rewrite history.
- You can always get back to where you started.
- Never, *ever, EVER* put sensitive information into a `git` repo. **ESPECIALLY** in public repos like GitHub.
- Read the above rule again.

Questions?

Hit me up on MacAdmins Slack, @nick.mcspadden
Hit me up on Twitter, @mrnickmcspadden