Crash course in version control (focusing mainly on *git*)

Cheng H. Lee
Lab7 Systems, Inc.

20 April 2016 — UT Biocomputing 2016

What is version control?

All “code” changes (bug fixes, improvements, etc.)

Need some way of managing changes; one naïve way:

```
my-script.py
my-script.py.0
my-script.py.1
my-script.py.2
```

-or-

```
my-script.py
my-script.py.2013-05-01
my-script.py.2012-12-20
my-script.py.2012-10-31
```

What is version control?

Many, many problems with the naïve approach:

- Requires needless duplication, clutters up filesystem
- Numbering scheme often delicate, hard to maintain
- Hard to understand history, relationships between versions and files.
- Hard to share and develop with multiple people

Most, if not all, of these problems solved by some sort of version control system (VCS).

What is version control *not?*

** A VCS is NO substitute for actual backups! **

Can help in recovering code/text (especially if distributed)…

…but most VCSes deal badly with large and/or binary files.

Also, I do *NOT* recommended using a VCS to manage:

- Large collections of binary files (e.g., PDFs)
- Large data files (e.g., genome references)

Basic VCS terminology

Repository: Some place that stores files, their past versions, and any associated metadata.

Working copy: Version of the repository currently being worked on, where changes to be added back to the repository are first produced.

Diff or **patch**: Description of how a specific file has changed.

Commit: Set of diffs and associated metadata (e.g., who made the change and when) that describe how the repository has changed from one version to another.

Lots of VCS out there

Centralized: single server storing the repository; all commits must be put onto this server.

E.g.: Subversion, CVS

Distributed: each developer has a copy of the repository; all commits happen "locally" but can be shared.

E.g.: Git, Mercurial

Also: Bzr, ClearCase, SourceSafe, RCS (not really...)

Basic centralized VCS workflow

1. Check out a working copy from VCS server.
2. Make changes in working copy.
3. Test changes to make sure they work.
4. Commit changes back to central server.
5. Repeat steps 2 through 4.

Basic distributed VCS workflow

Very similar...

1. Copy (or clone in git parlance) a repository.
2. Make changes in *your local copy*.
3. Test changes to make sure they work.
4. Commit changes to *your local copy*.
5. Repeat steps 2 through 4.

But we have the option of:

6. Sending our changes to someone else's repository, or
7. Pulling in changes from someone else's repository.

Getting started with git

Download and install:

Main page: http://git-scm.com/downloads

Windows: TortoiseGit (integrates with Explorer)

OS X: Use git-scm.com version (X Code version is old)

Debian/Ubuntu: "`apt-get install git`"

Minimal required configuration (tell git who you are):

```
$ git config user.name "first last"
$ git config user.email "me@institute.org"
```

Cloning a git repository

Cloning gets a repository from somewhere (e.g., GitHub), including all tracked files and their history.

```
# "git clone" will create a new subdirectory
# underneath your current location
$ cd $HOME/projects
$ ls
project1  project2
```

Cloning a git repository

Cloning gets a repository from somewhere (e.g., GitHub), including all tracked files and their history.

```
# Usage: "git clone <url>", where <url> is
# provided by person you're cloning from; e.g.,
$ git clone git@bitbucket.org:myorg/projectX.git
Cloning into 'projectX'
#  ... bunch of other status messages ...
```

Cloning a git repository

Cloning gets a repository from somewhere (e.g., GitHub), including all tracked files and their history.

```
$ ls
project1  project2  projectX
$ cd projectX
$ ls
# ... contents of the "projectX" repository ...
```

Setting up your own git repository

What if you have a project on your own computer that hasn't been shared with anyone else?

```
$ cd /path/to/my/project
$ ls -a
file1.txt  file2.txt  subdir/
$ git init
Initialized empty Git repository in /path/to/my/project/.git/
$ ls -a
.git/  file1.txt  file2.txt  subdir/
```

Where the git magic happens;
remove at your own peril

Adding files to version control

Git (and most other VCSes) do *not* automatically put files under version control.

Makes sense: don't want useless stuff (temporary files, large files, binary data, etc.) in the repository.

You must explicitly tell git what files you want to track.

Adding files to version control

What's in our project directory?

```
$ ls .
file1.txt  file2.txt  subdir/
$ ls subdir/
file3.txt  ignore-me.txt
```

Adding files to version control

```
# "git status": what's changed in your working directory
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# file1.txt
# file2.txt
# subdir/
nothing added to commit but untracked files present (use "git add" to track)
```

Adding files to version control

```
  # "git status": what's changed in your working directory
  $ git status
  # On branch master
  # Untracked files:
  #   (use "git add <file>..." to include in what will be
committed)
  #
  # file1.txt
  # file2.txt
  # subdir/
  nothing added to commit but untracked files present (use "git
add" to track)
```

"**untracked**": files git notices on your filesystem that are not yet under version control

Adding files to version control

```
  # "git status": what's changed in your working directory
  $ git status
  # On branch master
  # Untracked files:
  #   (use "git add <file>..." to include in what will be
committed)
  #
  # file1.txt
  # file2.txt
  # subdir/
  nothing added to commit but untracked files present (use "git
add" to track)
```

Note that subdirectory contents aren't listed; we'll come back to that in a bit.

Adding files to version control

```
  # "git status": what's changed in your working directory
  $ git status
  # On branch master
  # Untracked files:
  #   (use "git add <file>..." to include in what will be
committed)
  #
  # file1.txt
  # file2.txt
  # subdir/
  nothing added to commit but untracked files present (use "git
add" to track)
```

Git tells you exactly what to do

Adding files to version control

What's in our project directory?

```
$ ls .
file1.txt  file2.txt  subdir/
$ ls subdir/
file3.txt  ignore-me.txt
```

Adding files to version control

What's in our project directory?

```
$ ls .
file1.txt  file2.txt  subdir/
$ ls subdir/
file3.txt  ignore-me.txt
```

Let's say we only want to track file1.txt & file2.txt:

Adding files to version control

What's in our project directory?

```
$ ls .
file1.txt  file2.txt  subdir/
$ ls subdir/
file3.txt  ignore-me.txt
```

Let's say we only want to track file1.txt & file2.txt:

```
$ git add file1.txt
$ git add file2.txt
```

Adding files to version control

```
  $ git status
  # On branch master
  # Changes to be committed:
  #   (use "git rm --cached <file>..." to unstage)
  #
  #  new file:   file1.txt
  #  new file:   file2.txt
  #
  # Untracked files:
  #   (use "git add <file>..." to include in what will be
committed)
  #
  #  subdir/
```

Adding files to version control

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#  new file:   file1.txt
#  new file:   file2.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#  subdir/
```

"**staged**": git has detected changes, but hasn't saved ("**committed**") them yet.

Adding files to version control

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#  new file:    file1.txt
#  new file:    file2.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#  subdir/
```

in this case, two new files

Committing files to version control

```
# "git commit" puts stuff in the repository...
$ git commit -m "my first commit"
[master (root-commit) ec4107d] my first commit
1 files changed, 4 insertions(+), 0 deletions(-)
create mode 100644 file1.txt
create mode 100644 file2.txt
```

Committing files to version control

```
# "git commit" puts stuff in the repository...
$ git commit -m "my first commit"
[master (root-commit) ec4107d] my first commit
1 files changed, 4 insertions(+), 0 deletions(-)
create mode 100644 file1.txt
create mode 100644 file2.txt
```

commit message: tells people what you did

Committing files to version control

```
# "git commit" puts stuff in the repository...
$ git commit -m "my first commit"
[master (root-commit) ec4107d] my first commit
1 files changed, 4 insertions(+), 0 deletions(-)
create mode 100644 file1.txt
create mode 100644 file2.txt
```

SHA1 checksum: uniquely identifies commit; actually 40-characters long, but we can usually use just the 1st seven characters

What happens after the first commit?

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

Git tells us there's still stuff we aren't tracking.

Dealing with subdirectories

```
$ ls subdir/
file3.txt  ignore-me.txt
$ git add subdir
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#  new file:   subdir/file3.txt
#  new file:   subdir/ignore-me.txt
#
```

"git add <subdirectory name>"

Dealing with subdirectories

```
$ ls subdir/
file3.txt  ignore-me.txt
$ git add subdir
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#  new file:   subdir/file3.txt
#  new file:   subdir/ignore-me.txt
#
```

"git add <subdirectory name>"

adds all the files in the directory;
(might not be the desired behavior)

Dealing with subdirectories

```
  $ git add subdir/file3.txt
  $ git status
  # On branch master
  # Changes to be committed:
  #   (use "git reset HEAD <file>..." to unstage)
  #
  # new file:   subdir/file3.txt
  #
  # Untracked files:
  #   (use "git add <file>..." to include in what will be
committed)
  #
  #  subdir/ignore-me.txt
```

"git add <file name>"

Dealing with subdirectories

```
$ git add subdir/file3.txt
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   subdir/file3.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#  subdir/ignore-me.txt
```

"git add <file name>"

add just the file(s) you want
(don't forget to commit!)

Dealing with subdirectories

```
$ git add subdir/file3.txt
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   subdir/file3.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#  subdir/ignore-me.txt
```

"git add <file name>"

ignore all the others

Dealing with subdirectories

As a general rule,

```
  $ git <action> <subdirectory>
```

will apply said action to all files in the subdirectory.

When this is *not* what you want, you'll have to apply the action to each file individually:

```
  $ git <action> subdir/file_a
  $ git <action> subdir/file_b
  $ ......
```

Ignoring certain files

Having files show up as "untracked" all the time can be annoying. Use the **.gitignore** file to ignore them:

```
$ cd /path/to/my/project
$ ls -a
.git/  file1.txt  file2.txt  subdir/
```

easiest to put it where your repository's ".git" directory is

Ignoring certain files

Having files show up as "untracked" all the time can be annoying. Use the **.gitignore** file to ignore them:

```
$ cd /path/to/my/project
$ ls -a
.git/  file1.txt  file2.txt  subdir/
$ echo "subdir/ignore-me.txt" > .gitignore
$ echo ".*.swp" >> .gitignore
$ echo "*~"     >> .gitignore
```

Ignoring certain files

Having files show up as "untracked" all the time can be annoying. Use the **.gitignore** file to ignore them:

```
$ cd /path/to/my/project
$ ls -a
.git/  file1.txt  file2.txt  subdir/
$ echo "subdir/ignore-me.txt" > .gitignore
$ echo ".*.swp" >> .gitignore
$ echo "*~"     >> .gitignore
```

ignore specific source files

Ignoring certain files

Having files show up as "untracked" all the time can be annoying. Use the **.gitignore** file to ignore them:

```
$ cd /path/to/my/project
$ ls -a
.git/  file1.txt  file2.txt  subdir/
$ echo "subdir/ignore-me.txt" > .gitignore
$ echo ".*.swp" >> .gitignore
$ echo "*~"     >> .gitignore
```

things like editor temp. files

Ignoring certain files

".gitignore" is a regular text file.

You can edit it with any text editor.

```
$ nano .gitignore
# ... add “.*pyc” as a new line to have git
# ignore compiled python files ...
```

You can add it to version control.

```
# useful for multi-person projects
$ git add .gitignore
$ git commit -m "added a .gitignore file"
... info about the commit ...
```

Adding more files to the repository

```
# Create a new file; hopefully, you're doing
# something a little more impressive.
$ echo "hello world" > subdir/file4.txt
```

Adding more files to the repository

```
# Create a new file; hopefully, you're doing
# something a little more impressive.
$ echo "hello world" > subdir/file4.txt

$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       subdir/file4.txt
nothing added to commit but untracked files present (use "git add" to track)
```

Adding more files to the repository

Follow the standard approach:

```
$ git add subdir/file4.txt
$ git commit -m "added file4.txt"
[master 1fede62] added file4.txt
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 subdir/file4.txt
```

What's happened to our code?

To get a history of commits to your repository:

```
$ git log
commit 1fede6267aaa964995f722f8aa5503cd390f946e
Author: Cheng H. Lee <cheng.lee@lab7.io>
Date:   Thu May 2 19:35:32 2013 -0500

added file4.txt

commit 3e36430d2a9d519897e5c6f7e1922a31e3ab4d14
Author: Cheng H. Lee <cheng.lee@lab7.io>
Date:   Thu May 2 19:21:22 2013 -0500

added a .gitignore file

... and so on ...
```

What's happened to our code?

To get a history of commits to your repository:

```
$ git log
commit 1fede6267aaa964995f722f8aa5503cd390f946e
Author: Cheng H. Lee <cheng.lee@lab7.io>
Date:   Thu May 2 19:35:32 2013 -0500

added file4.txt

commit 3e36430d2a9d519897e5c6f7e1922a31e3ab4d14
Author:
Date:   Thu May 2 19:21:22 2013 -0500

added a .gitignore file

... and so on ...
```

The most recent commit...

What's happened to our code?

To get a history of commits to your repository:

```
$ git log
commit 1fede6267aaa964995f722f8aa5503cd390f946e
Author: Cheng H. Lee <cheng.lee@lab7.io>
Date:   Thu May 2 19:35:32 2013 -0500

added file4.txt

commit 3e36430d2a9d519897e5c6f7e1922a31e3ab4d14
Author: Cheng H. Lee <cheng.lee@lab7.io>
Date:   Thu May 2 19:21:22 2013 -0500

added a .gitignore file

... and so on ...
```

...and the one before that

What's happened to our code?

"git log" has lots of options:

```
$ git log -5            # only the last 5 commits
... as before, but we'll only get 5 messages ...

$ git log --oneline     # abbreviated log
1fede62 added file4.txt
3e36430 added a .gitignore file
3212151 added file3.txt
ec4107d my first commit

$ git log -- file1.txt  # show commits involving file1.txt

$ git help log          # bring up help page for more options
```

Committing edits to the repository

Let's say I've just finished editing "file1.txt".

```
  $ 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:   file1.txt
  #
  no changes added to commit (use "git add" and/or "git commit -
a")
```

git has detected that the file has changed.

Committing edits to the repository

To figure out what has changed since the last commit:

```
$ git diff --color
```

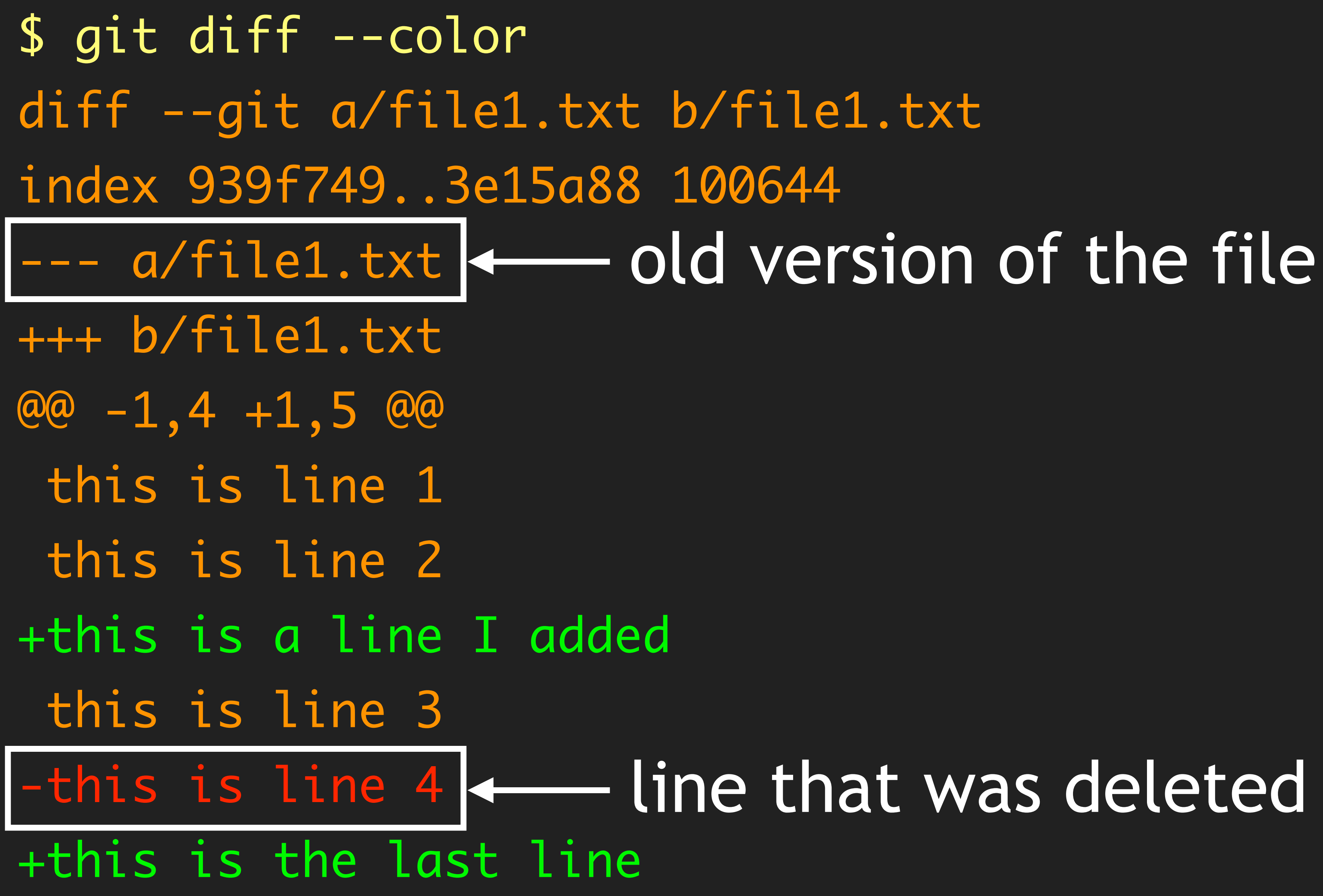
Committing edits to the repository

Output in “unified diff” (AKA “patch”) format

```
$ git diff --color
diff --git a/file1.txt b/file1.txt
index 939f749..3e15a88 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,4 +1,5 @@
 this is line 1
 this is line 2
+this is a line I added
 this is line 3
-this is line 4
+this is the last line
```

Committing edits to the repository

Output in “unified diff” (AKA “patch”) format



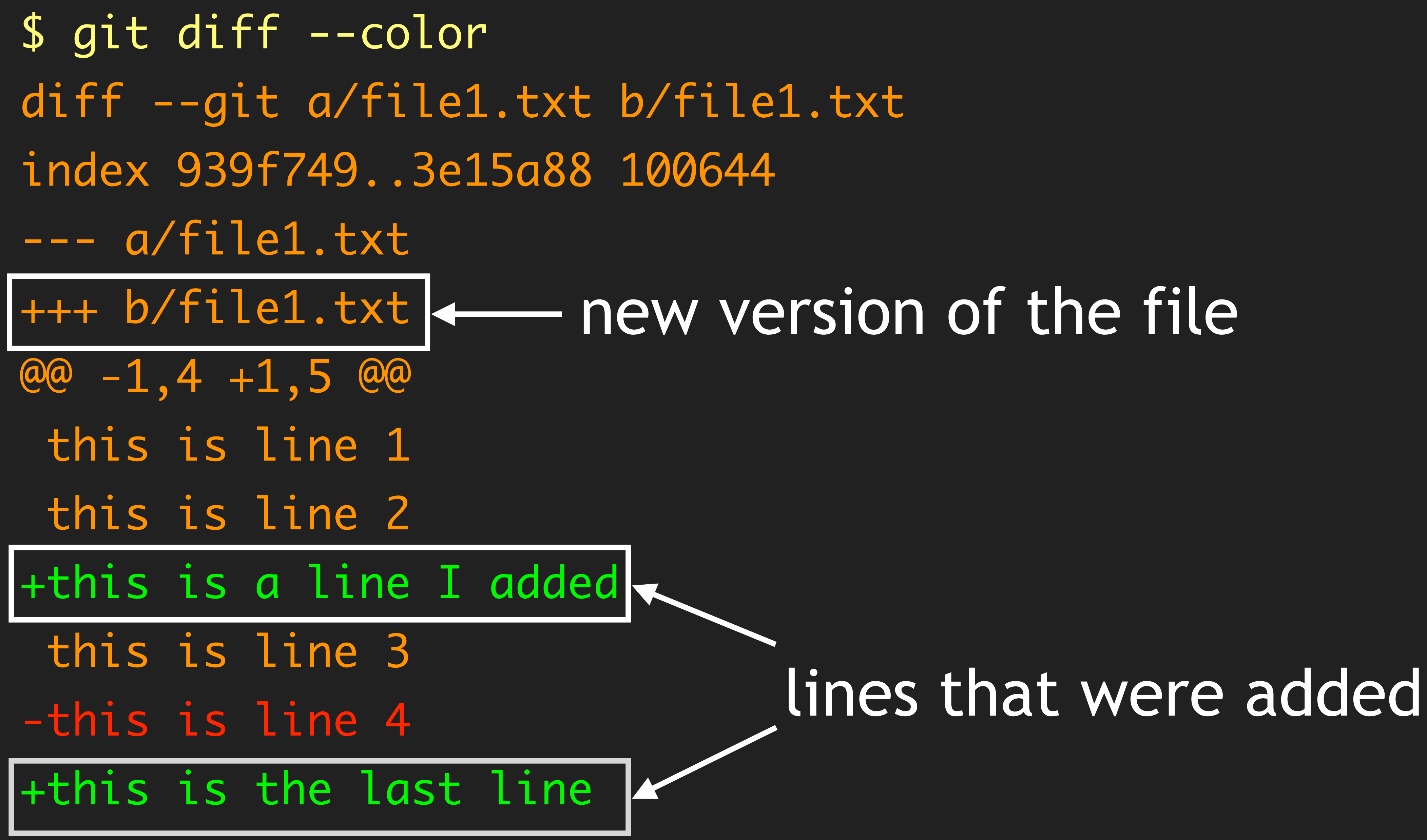
```
$ git diff --color
diff --git a/file1.txt b/file1.txt
index 939f749..3e15a88 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,4 +1,5 @@
 this is line 1
 this is line 2
+this is a line I added
 this is line 3
-this is line 4
+this is the last line
```

old version of the file

line that was deleted

Committing edits to the repository

Output in “unified diff” (AKA “patch”) format



```
$ git diff --color
diff --git a/file1.txt b/file1.txt
index 939f749..3e15a88 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,4 +1,5 @@
 this is line 1
 this is line 2
+this is a line I added
 this is line 3
-this is line 4
+this is the last line
```

new version of the file

lines that were added

Committing edits to the repository

VCSes don't record changes until you **commit**.

Unlike other VCSes, git "requires" a two-step commit:

```
$ git add file1.txt    # "stages" file1
$ git commit -m "edits made to file1"
[master 51cb5a3] edits made to file1
1 files changed, 2 insertions(+), 1 deletions(-)
```

If you forget to **stage** a file with "git add", "git commit" won't actually commit its changes into the repository.

Committing edits to the repository

There is a short-cut for the lazy. Suppose:

```
$ 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:   file2.txt
#       modified:   subdir/file3.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Committing edits to the repository

The "long" way of committing both files:

```
$ git add file2.txt subdir/file3.txt
$ git commit -m "Changes to file2 and file3"
[master 0724984] changes to file2 and file3
2 files changed, 5 insertions(+), 0 deletions(-)
```

Committing edits to the repository

“Shorter” way: “git add -u” to stages all modified files.

```
$ git add -u
$ git status
# On branch master
# Changes to be committed:
#       modified:   file2.txt
#       modified:   subdir/file3.txt
# ... other status messages ...
$ git commit -m "Changes to file2 and file3"
[master 0724984] changes to file2 and file3
2 files changed, 5 insertions(+), 0 deletions(-)
```

Committing edits to the repository

“Shortest” way of committing updated files:

```
$ git commit -a -m "Changes to file2 and file3"
[master 0724984] changes to file2 and file3
2 files changed, 5 insertions(+), 0 deletions(-)
```

“git commit -a”: “stage all tracked files that have been modified and then commit them”.

This mimics the "commit" behavior of other VCSes.

Committing edits to the repository

Caveat: "git commit -a" does not automatically add untracked files to the commit. If you create a new file, you must explicitly use "git add" to commit it.

E.g., say you modified "file2.txt" and "file3.txt" and added a new file called "useful-code.py". To commit all three, you *must* run the following:

```
$ git add useful-code.py
$ git commit -a -m "my commit message"
[master 4f9a57f] my commit message
2 files changed, 5 insertions(+), 0 deletions(-)
create mode 100644 useful-code.py
```

Removing files

Occasionally useful to remove files from your working copy; e.g., old code that conflicts with your new code:

```
$ ls
file1.txt  file2.txt  old-script.py  subdir/
$ git rm old-script.py
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#     deleted:    old-script.py
#
```

staged but doesn't take effect until commit.

Removing files

Occasionally useful to remove files from your working copy; e.g., old code that conflicts with your new code:

```
$ git commit -m "removed obsolete script"
[master 9458cbb] removed obsolete script
1 files changed, 0 insertions(+), 4 deletions(-)
delete mode 100644 old-script.py
$ ls
file1.txt  file2.txt  subdir/
```

"old-script.py" no longer exists in the directory.

Moving or renaming files

Often need to move or rename files:

```
$ git mv file2.txt subdir/new-name.txt
# As with "git rm", this stages but does not commit the file.
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      renamed:    file2.txt -> subdir/new-name.txt
#
$ git commit -m "renamed file2.txt to subdir/new-name.txt"
$ ls subdir
new-name.txt
```

Dead but not fogotten

Why use a VCS?

Once something is in the repository, it is never lost.*

Among other things, we can:

- Save ourselves from some types of “rm” trouble.
- Compare any two previous (committed) versions.
- Backing out from recent changes.
- Bring back a file from the dead.

* Well, unless the entire repository itself (i.e., the ".git" directory) is lost.**
** Or with git, you do something really bad like rebase a public branch, then run gc.

Saving yourself from trouble

Commonly, trigger happiness with "rm":

```
$ ... do some work ...
$ ls
file1.txt  file2.txt  file_a.txt  file_b.txt  subdir/

# "file_a.txt" and "file_b.txt" were generated as temporary
# files while I was doing work; don't need them any more...
$ rm -f file*

# OOPS!
$ ls
subdir/
```

Saving yourself from trouble

After deleting files:

```
  $ git status
  # On branch master
  # Changes not staged for commit:
  #   (use "git add/rm <file>..." to update what will be
committed)
  #   (use "git checkout -- <file>..." to discard changes in
working directory)
  #
  #       deleted:    file1.txt
  #       deleted:    file2.txt
  #
  no changes added to commit (use "git add" and/or "git commit -
a")
```

← Important files I cared about

Saving yourself from trouble

After deleting files:

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    file1.txt
#       deleted:    file2.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Follow the instructions to "recover"

Saving yourself from trouble

Recovering files from the repository:

```
$ git checkout -- file1.txt file2.txt
$ ls
file1.txt  file2.txt  subdir/
```

Saving yourself from trouble

Recovering files from the repository:

```
$ git checkout -- file1.txt file2.txt
$ ls
file1.txt  file2.txt  subdir/
```

Important caveat: *"git checkout" can only recover files up to the last commit. All uncommitted changes are permanently destroyed by “rm”.*

Looking at/comparing to previous commits

Two main tools to look at old versions (commits):

- git log: fetch the previous commit logs and metadata
- git diff: generate a diff between two commits

```
# "git log" general command format
$ git log <options> <since commit>..<until commit> -- <files>

# "git diff" general command format
$ git diff <options> <since commit> -- <files>
```

git has multiple ways of referring commits; the "--" is a way of saying everything after this is the name of a file, not the name of a commit

How git refers to commits

Two more common ways:

<SHA1 checksum>: Absolute & unambiguous way

<commit>~<N>: <N>th-generation ancestor of commit

But there are many other ways; see "git help revisions".

"**HEAD**": Special name referring to the last commit*

"git status": compare current state to HEAD

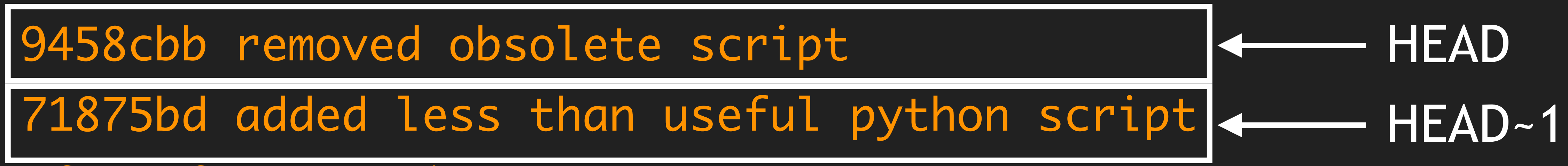
"HEAD~5": 6 commits ago

* "last commit from where you are now, which might not be the latest commit."

How git refers to commits

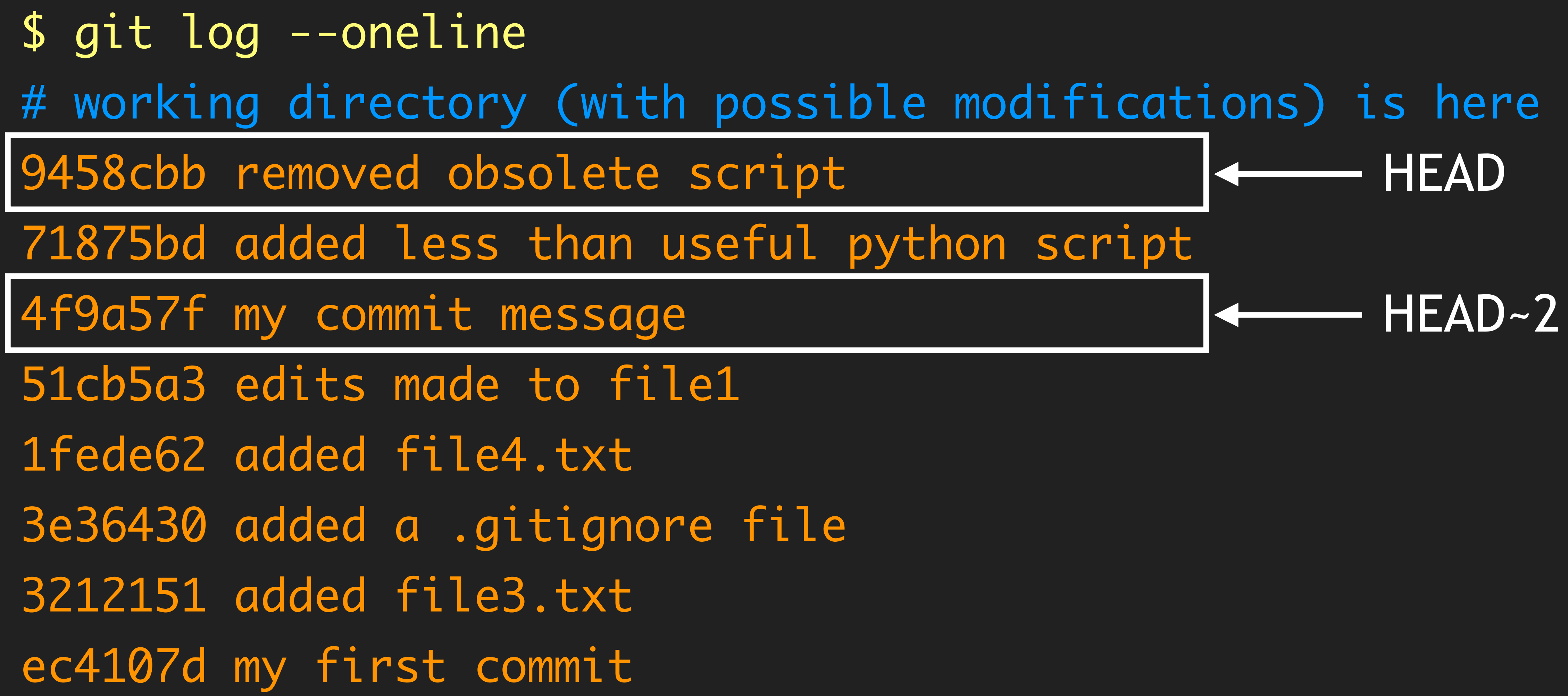
```
$ git log --oneline
# working directory (with possible modifications) is here
9458cbb removed obsolete script             ← HEAD
71875bd added less than useful python script
4f9a57f my commit message
51cb5a3 edits made to file1
1fede62 added file4.txt
3e36430 added a .gitignore file
3212151 added file3.txt
ec4107d my first commit
```

How git refers to commits



```
$ git log --oneline
# working directory (with possible modifications) is here
9458cbb removed obsolete script
71875bd added less than useful python script
4f9a57f my commit message
51cb5a3 edits made to file1
1fede62 added file4.txt
3e36430 added a .gitignore file
3212151 added file3.txt
ec4107d my first commit
```

How git refers to commits



```
$ git log --oneline
# working directory (with possible modifications) is here
9458cbb removed obsolete script                        HEAD
71875bd added less than useful python script
4f9a57f my commit message                              HEAD~2
51cb5a3 edits made to file1
1fede62 added file4.txt
3e36430 added a .gitignore file
3212151 added file3.txt
ec4107d my first commit
```

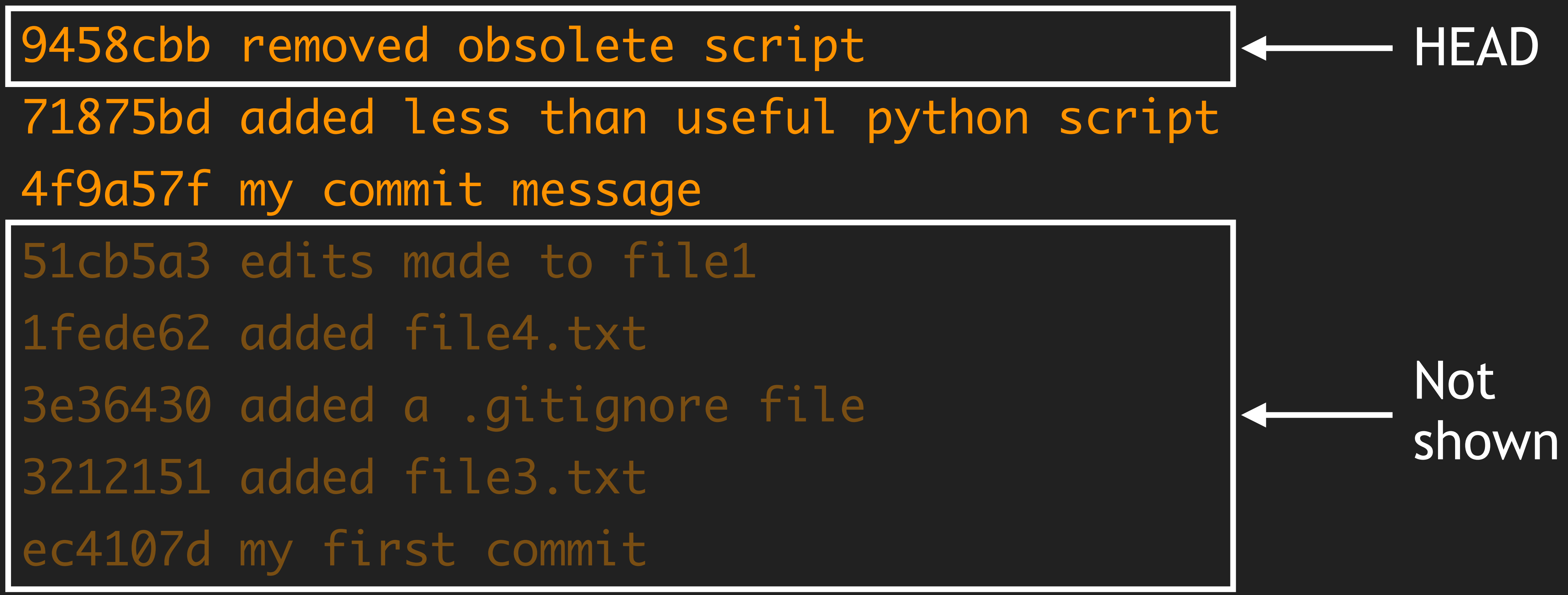
How git refers to commits

```
$ git log --oneline
# working directory (with possible modifications) is here
9458cbb removed obsolete script                  ← HEAD
71875bd added less than useful python script
4f9a57f my commit message
51cb5a3 edits made to file1
1fede62 added file4.txt
3e36430 added a .gitignore file                  ← HEAD~5
3212151 added file3.txt
ec4107d my first commit
```

How git refers to commits

What's changed in the repository since 4 commits ago?

```
# "git log" is not inclusive of the <since> commit.
# Also, if we leave off a commit reference, git assumes
# "HEAD"; so, these two are the same command:
$ git log --oneline HEAD~3..HEAD
$ git log --oneline HEAD~3..
```



```
9458cbb removed obsolete script
71875bd added less than useful python script
4f9a57f my commit message
51cb5a3 edits made to file1
1fede62 added file4.txt
3e36430 added a .gitignore file
3212151 added file3.txt
ec4107d my first commit
```

HEAD

Not shown

How git refers to commits

Relative references (~<N>) are for commits, not files.

```
$ git log --oneline -- file1.txt
51cb5a3 edits made to file1   ← HEAD~3
ec4107d my first commit       ← HEAD~7
```

```
# What's changed in file1.txt in the last 2 commits?
$ git log --oneline HEAD~2..
9458cbb removed obsolete script
71875bd added less than useful python script
$ git log --oneline HEAD~2.. -- file1.txt
$                                             ← No output since
                                                nothing changed
                                                in file1.txt
```

Quick word about commit logs

So far, we've been using "`commit -m 'one line message'`" to generate our commit logs.

Better practice for commits is:

```
$ git commit -a
... Brings up a text editor for you to enter a log message ...
```

This allows you to provide more informative messages.

Six months from now, you'll appreciate it.

Quick word about commit logs

De-facto community standard for log message.

```
First line: short description of what was changed (<50 chars)
# --- empty second line ---
Multiple lines providing more details about what was
changed (e.g., what algorithm was implemented), and
more importantly, why it was changed.
Often wrapped to 72 characters per line.
```

Quick word about commit logs

Example from one of my projects:

```
$ git log -1 b09eee9
commit b09eee938ce52b35026972b76897086c992145a2
Author: Cheng H. Lee <cheng.lee@lab7.io>
Date:   Mon Apr 29 13:22:32 2013 -0500

CORE-258 mutation detection for JSONHstore by default

Made SQLAlchemy mutation detection and notification
the default behavior for JSONHstore; fixed problems we've
had with multiple JSON-encoding passes by using the prefix
tagging trick used with JSONArray (commit 7728c56).
```

Quick word about commit logs

Example from one of my projects:

```
$ git log -1 b09eee9
commit b09eee938ce52b35026972b76897086c992145a2
Author: Cheng H. Lee <cheng.lee@lab7.io>
Date:   Mon Apr 29 13:22:32 2013 -0500

[illegible] fault

Made SQLAlchemy mutation detection and notification
the default behavior for JSONHstore; fixed problems we've
had with multiple JSON-encoding passes by using the prefix
tagging trick used with JSONArray (commit 7728c56).
```

Metadata: commit id, who, when

Quick word about commit logs

Example from one of my projects:

```
$ git log -1 b09eee9
commit b09eee938ce52b35026972b76897086c992145a2
Author: Cheng H. Lee <cheng.lee@lab7.io>
Date:   Mon Apr 29 13:22:32 2013 -0500

CORE-258 mutation detection for JSONHstore by default
```

Short description: bug id, what was fixed
What shows up when we do "git log --oneline"

```
've
efix
tagging trick used with JSONArray (commit 7728c56).
```

Quick word about commit logs

Example from one of my projects:

```
$ git log -1 b09eee9
commit b09eee938ce52b35026972b76897086c992145a2
Author: Cheng H. Lee <cheng.lee@lab7.io>
```

Gory details: why I fixed it, algorithm used, where the fix idea come from, etc.

```
CORE-258 mutation detection for JSONHstore by default
```

```
Made SQLAlchemy mutation detection and notification
the default behavior for JSONHstore; fixed problems we've
had with multiple JSON-encoding passes by using the prefix
tagging trick used with JSONArray (commit 7728c56).
```

Comparing to older versions

What have I changed since the last commit?

```
$ echo "this is the new last line" >>file1.txt

# git diff compares your edited version with some commit
# Implicitly, this is HEAD. So, these are equivalent:
$ git diff -- file1.txt
$ git diff HEAD -- file1.txt
diff --git a/file1.txt b/file1.txt
index 3721789..e77d501 100644
--- a/file1.txt
+++ b/file1.txt
@@ -3,3 +3,5 @@
... rest of diff output ...
```

Comparing to older versions

Can also get a single diff against any previous version

```
$ git log --oneline -- file1.txt
51cb5a3 edits made to file1
ec4107d my first commit
```

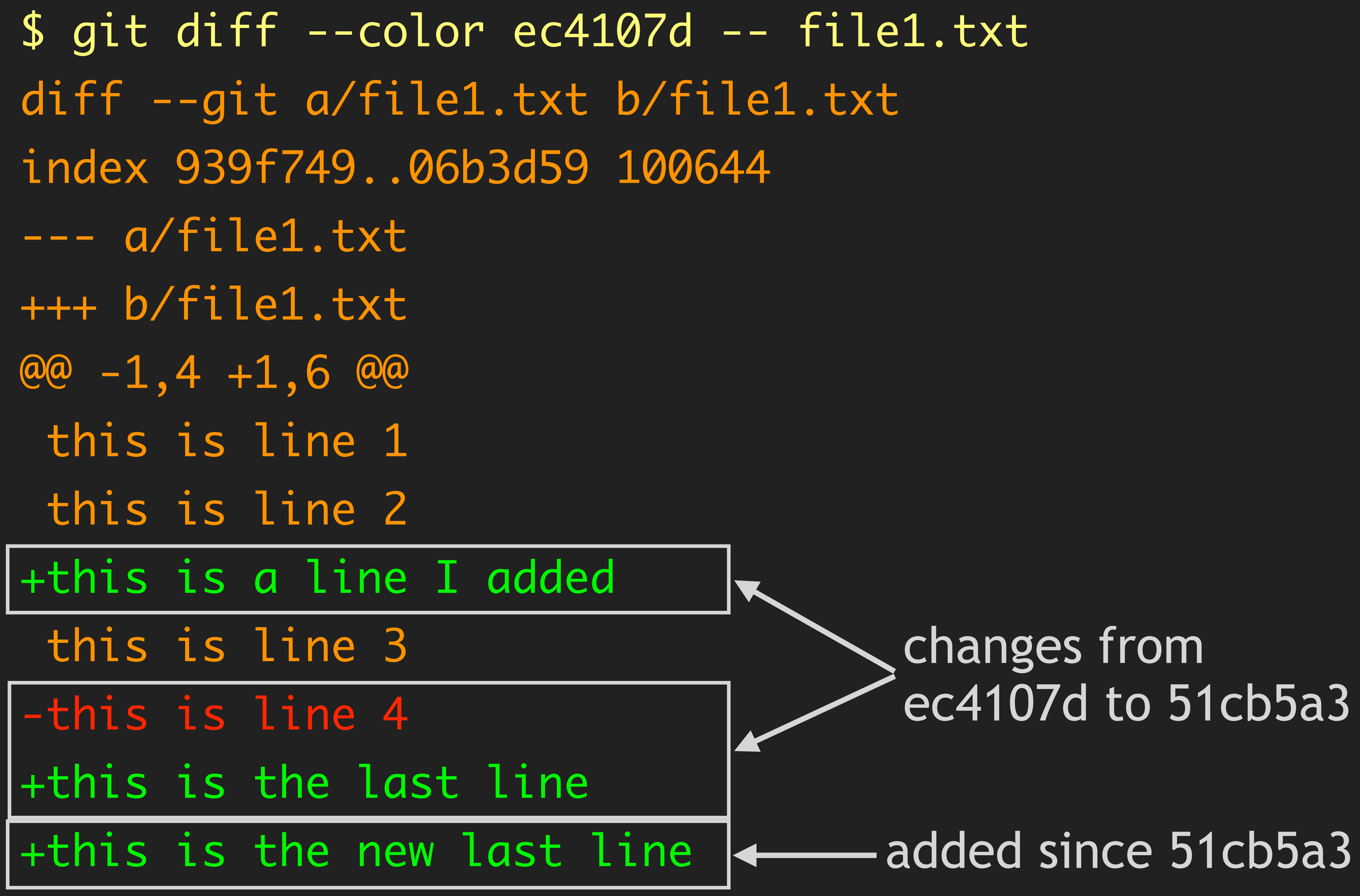
Comparing to older versions

Can also get a single diff against any previous version

```
$ git diff --color 51cb5a3 -- file1.txt
diff --git a/file1.txt b/file1.txt
index 3721789..06b3d59 100644
--- a/file1.txt
+++ b/file1.txt
@@ -3,3 +3,4 @@ this is line 2
 this is a line I added
 this is line 3
 this is the last line
+this is the new last line
```

Comparing to older versions

Can also get a single diff against any previous version



```
$ git diff --color ec4107d -- file1.txt
diff --git a/file1.txt b/file1.txt
index 939f749..06b3d59 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,4 +1,6 @@
 this is line 1
 this is line 2
+this is a line I added
 this is line 3
-this is line 4
+this is the last line
+this is the new last line
```

changes from
ec4107d to 51cb5a3

added since 51cb5a3

Bringing back an old version

Suppose you realize the old version of a file was better:

```
$ git log --oneline -- file1.txt
51cb5a3 edits made to file1
ec4107d my first commit
$ git checkout ec4107d -- file1.txt
$ cat file1.txt
# ... should see the contents of ec4107d here ...
```

***Warning**: This will silently and irrevocably destroy any changes you've made to "file1.txt" since its last commit!*

Bringing back an old version

Checkout only stages the file:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   file1.txt
#
$ git commit -m "restored original version of file1"
```

Old version won't be fully restored in the repository until the actual commit.

"checkout" to undelete a file

Can use checkout to restore a file deleted by "git rm":

```
# Use "git log" to find the commit that deleted the file
#   "--diff-filter=D": look for commits that deleted a file
#   "-1": show only the last relevant commit
$ git log --diff-filter=D -1 --oneline -- old-script.py
9458cbb removed obsolete script

# Need to go back one commit (~1) so the file exists...
$ git checkout 9458cbb~1 -- old-script.py

$ git commit -m "restored my old python script"
```

Be careful with checkout!

Make sure you supply "-- <filename>"; without it:

```
$ git checkout ec4107d
... Warning about 'detached HEAD' state ...
```

Rolls your working directory & all files back to their state in the specified commit (probably not what you want).

To get out of this situation:

```
$ git checkout --force master
```

Getting the contents of an old version

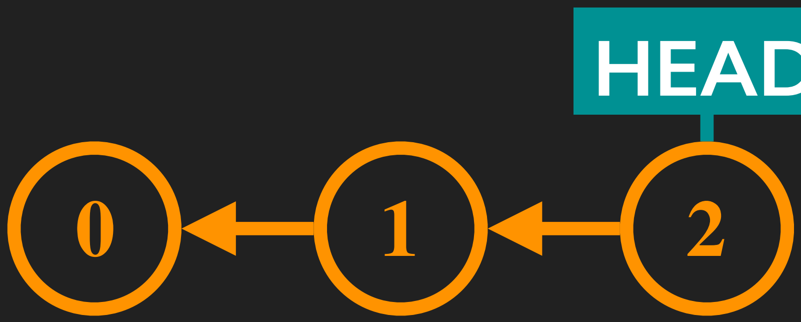
Sometimes, we just want to see the contents of an old version of a file (without restoring in the repository):

```
# Dump the contents to the terminal
$ git show <commit>:my-old-file.txt
# Dump the contents to a file named "new-file.txt"
$ git show <commit>:my-old-file.txt > new-file.txt

# <commit> can be any valid commit reference; e.g.,
$ git show HEAD~1:file1.txt   # relative to last commit
$ git show 51cb5a3:file1.txt  # absolute commit identifier
```

The “HEAD” commit

git tracks history as directed, acyclic graph of commits.

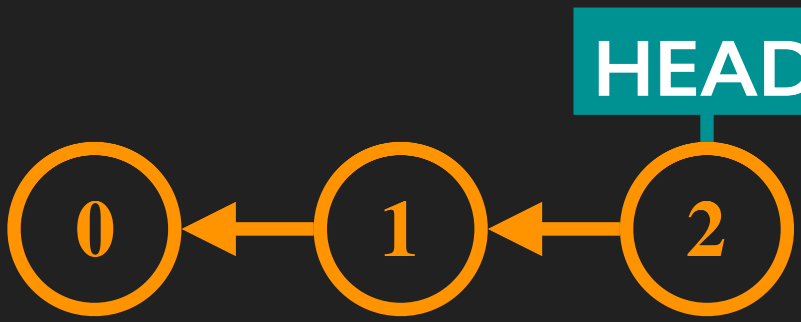


Each commit knows who its parent(s) is/are.

HEAD: special label for the commit you’re working from; i.e., all changes when you run “git status”, “git diff”, etc. are determined relative to HEAD.

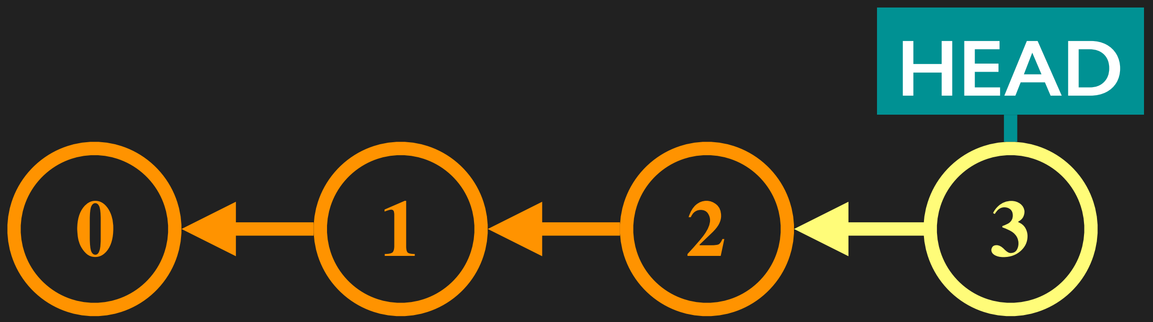
The “HEAD” commit

Committing moves HEAD.



The “HEAD” commit

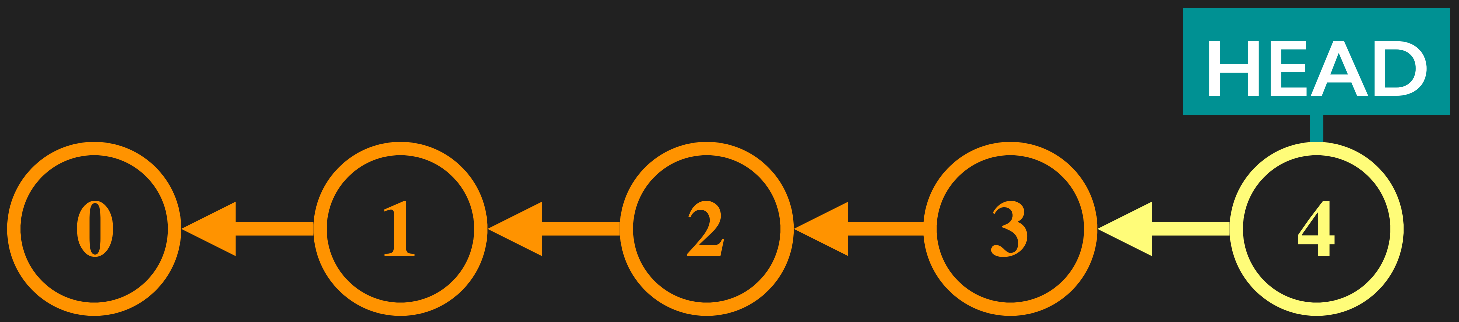
Committing moves HEAD.



Running “`git commit`”…

The "HEAD" commit

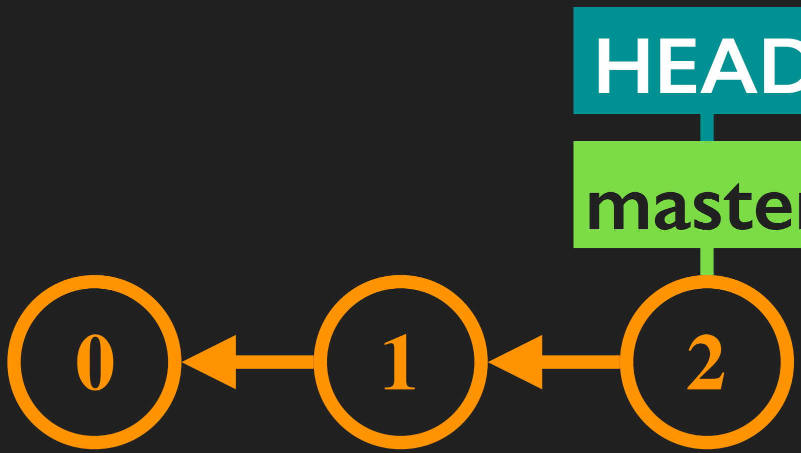
Committing moves HEAD.



Running "`git commit`"…again…

Branches

Normally, we don’t directly refer to “HEAD”.

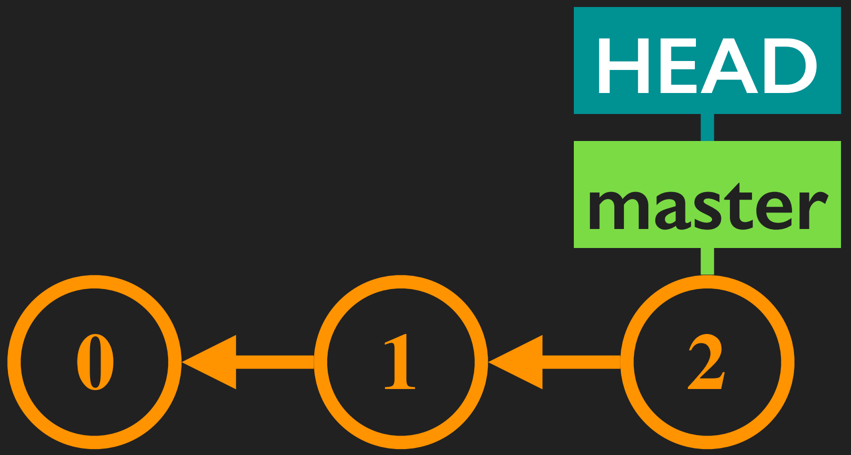


Instead, we attach “HEAD” to another type of label called a “**branch**” and refer to that branch.

*Git automatically creates a branch called “**master**” when a repository is first created.*

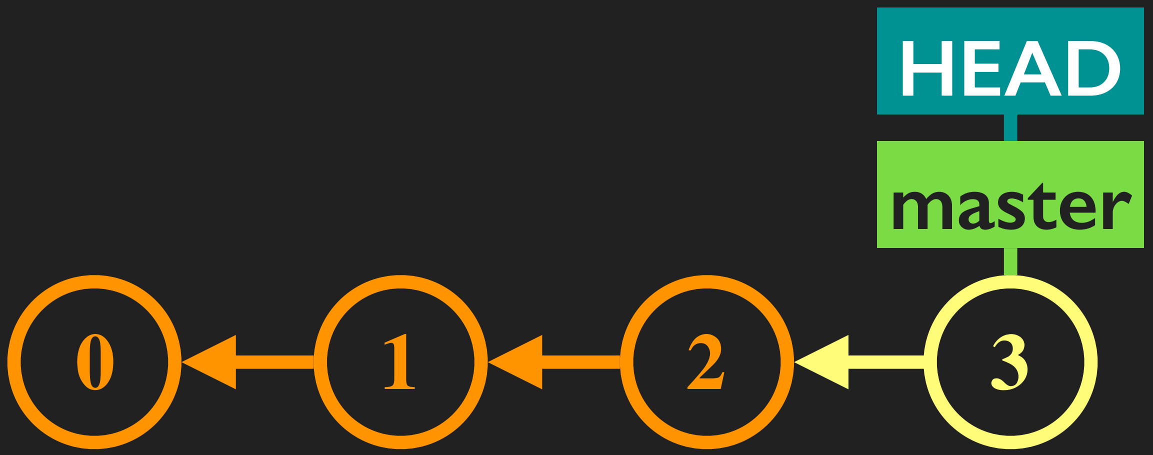
Committing on branches

Committing advances both HEAD and the branch (tip)



Committing on branches

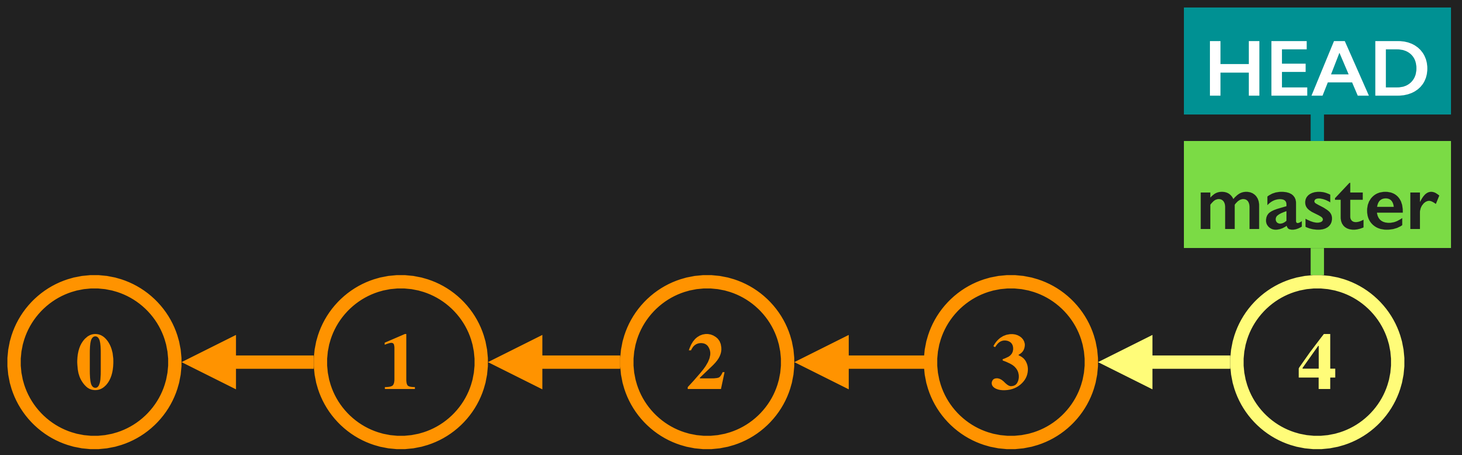
Committing advances both HEAD and the branch (tip).



Running “`git commit`”…

Committing on branches

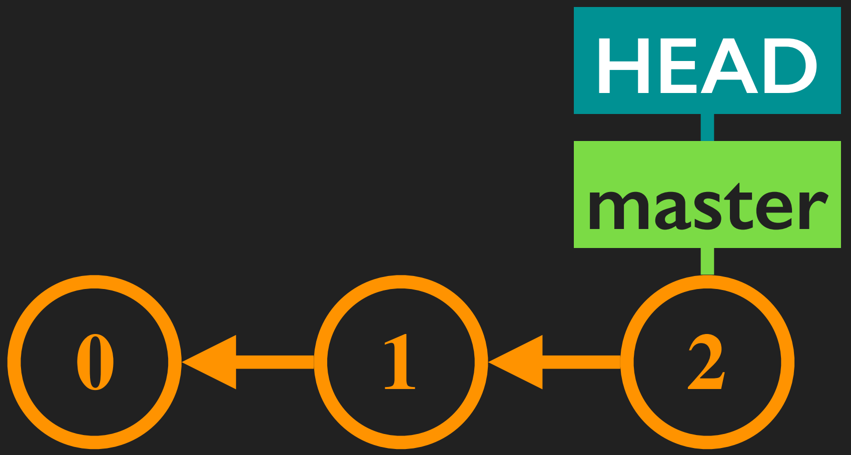
Committing advances both HEAD and the branch (tip).



Running “`git commit`”…again…

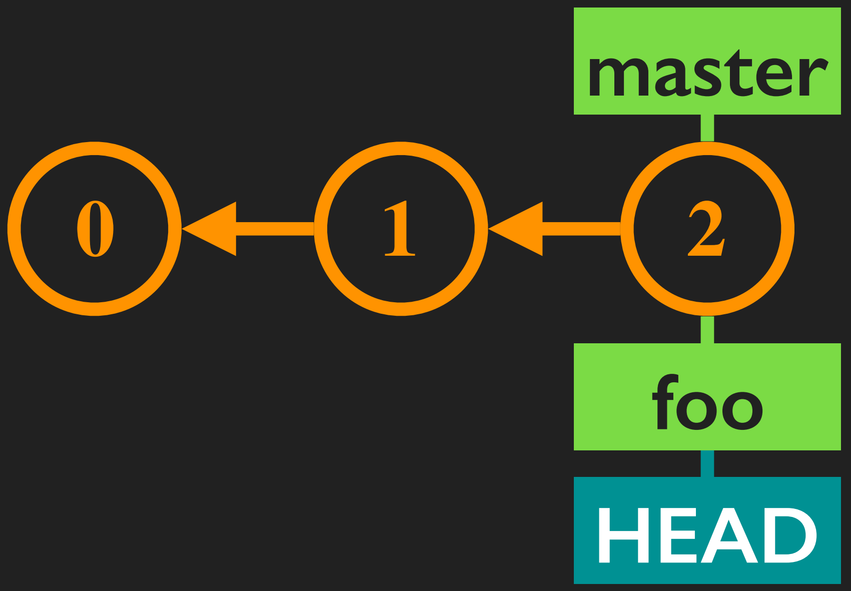
Creating new branches

Git makes it easy to create new branches.



Creating new branches

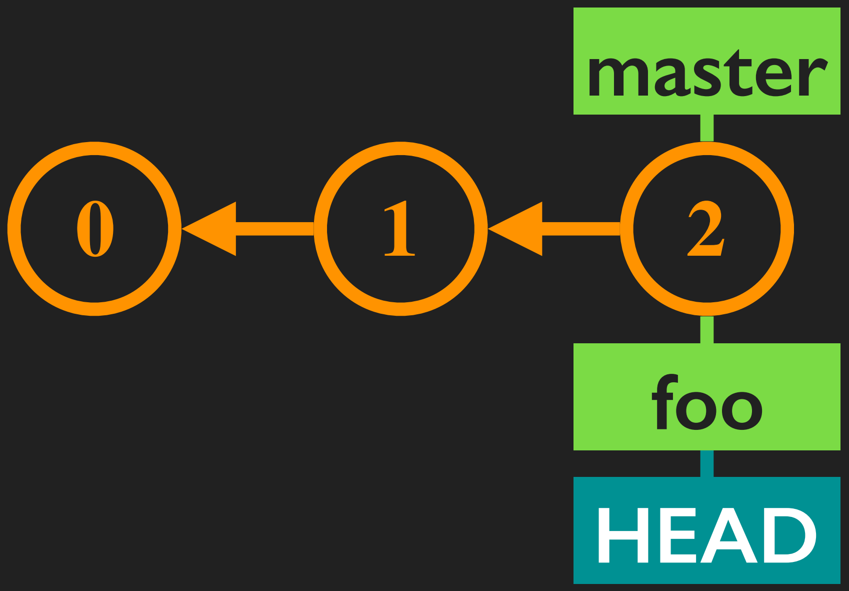
Git makes it easy to create new branches.



"`git checkout -b foo`": create a new branch called "foo" and set HEAD to follow it.

Committing on a branch

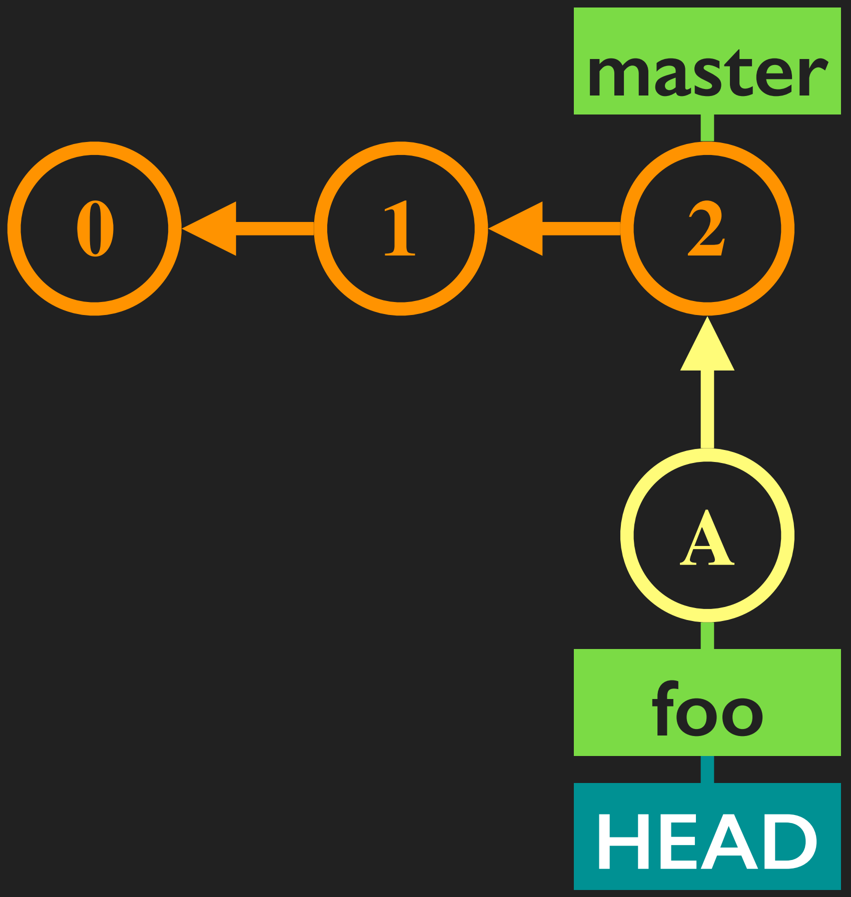
Commits advance HEAD and the current branch.



Starting on the “foo” branch (i.e., where HEAD is attached)…

Committing on a branch

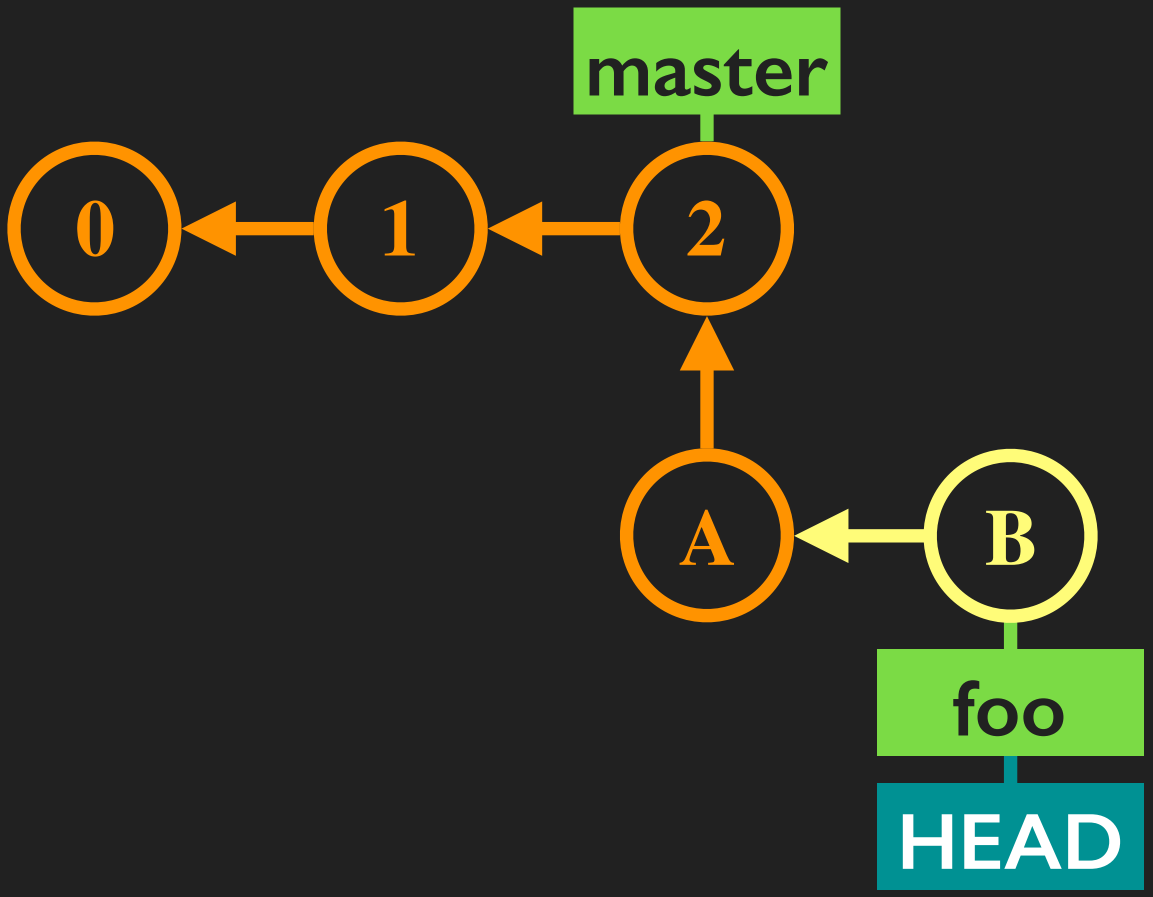
Commits advance HEAD and the current branch.



Running “`git commit`”…

Committing on a branch

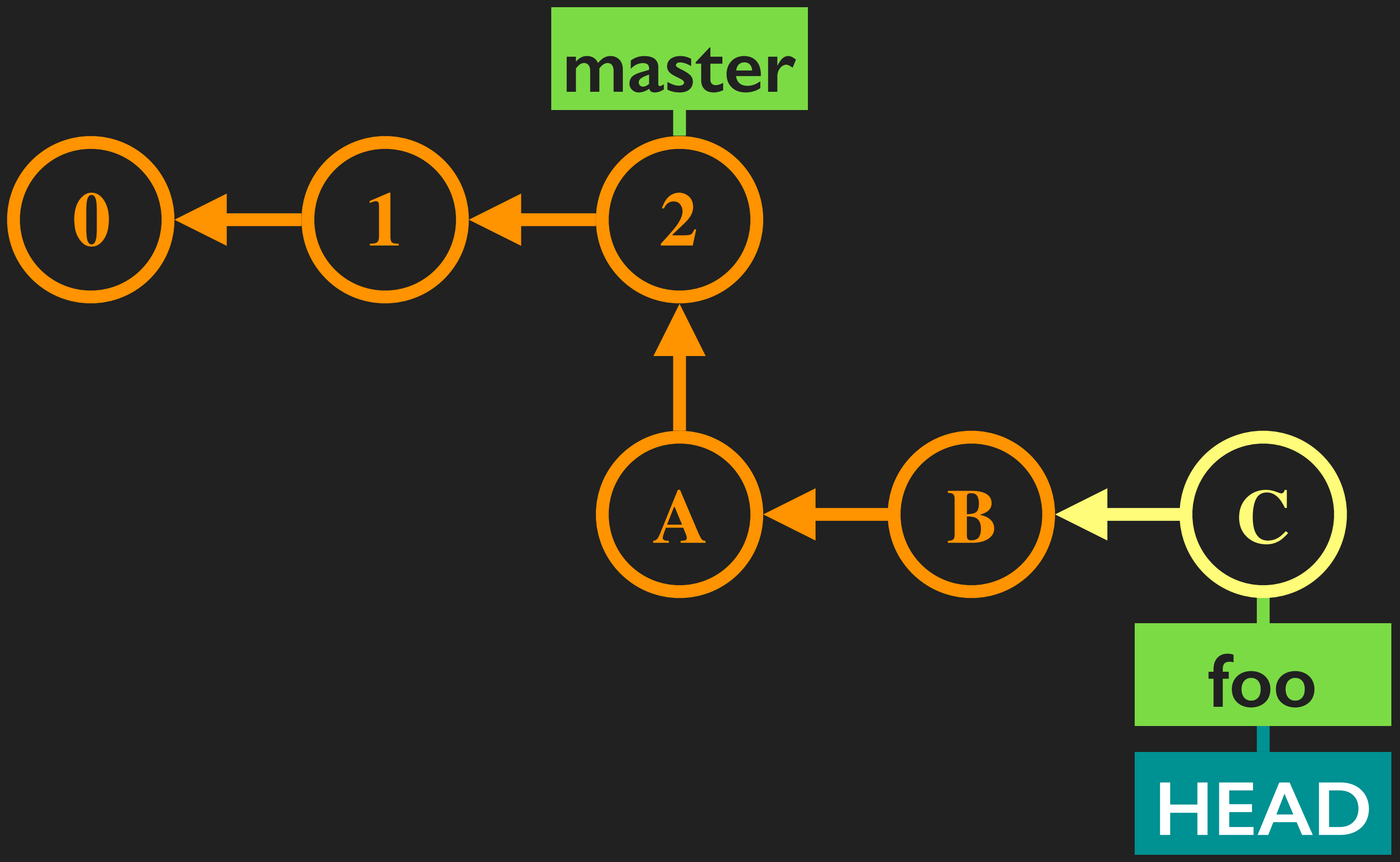
Commits advance HEAD and the current branch.



Running “git commit”…again…

Committing on a branch

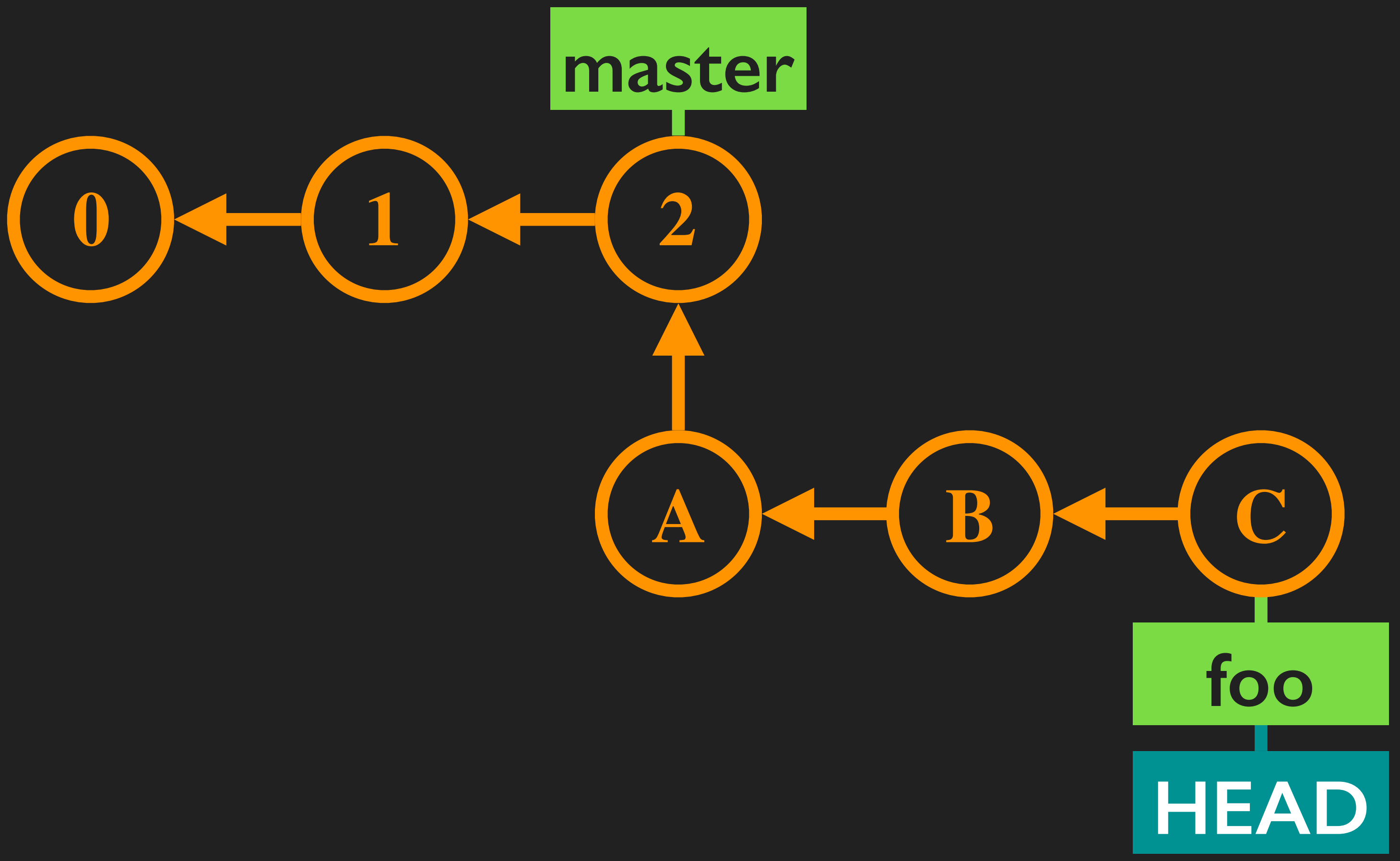
Commits advance HEAD and the current branch.



Running “`git commit`”...again...and again.

Switching branches with “checkout”

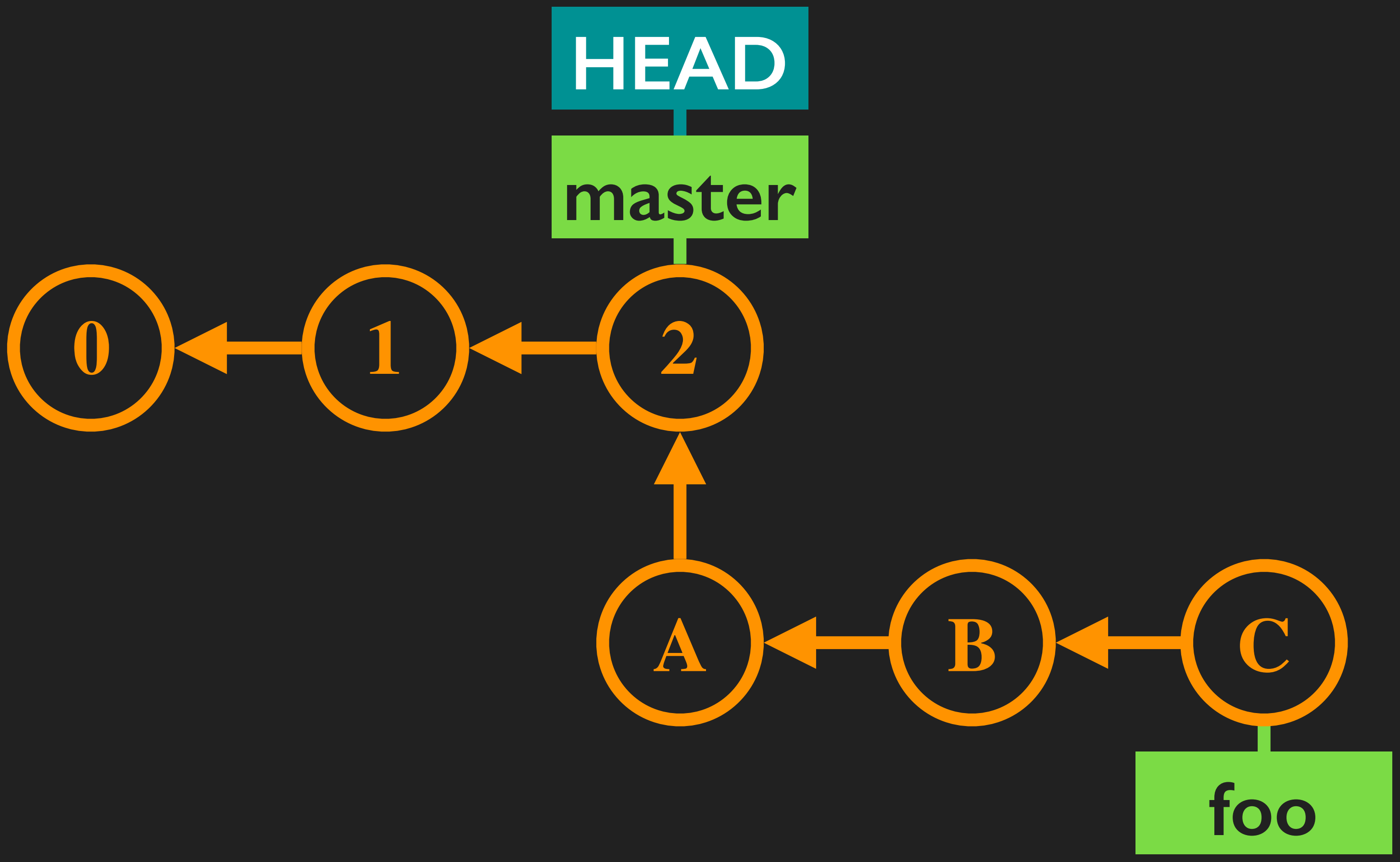
“`checkout`” changes which branch HEAD is attached to.



Starting from branch “foo”…

Switching branches with “checkout”

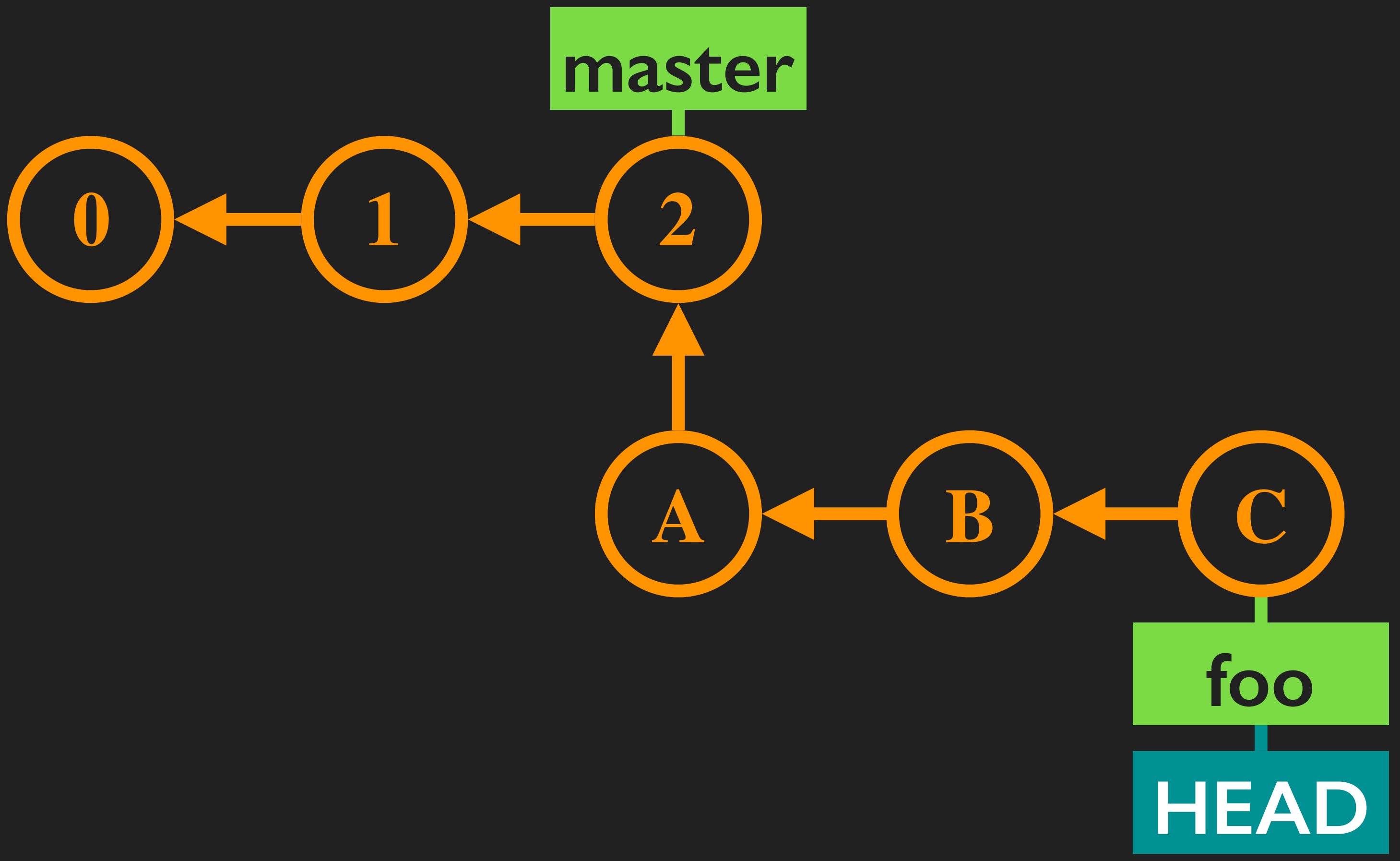
“`checkout`” changes which branch HEAD is attached to.



…running “`git checkout master`” switches back to the “master” branch.

Switching branches with "checkout"

"`checkout`" changes which branch HEAD is attached to.



Running "`git checkout foo`" switches us from the "master" branch back onto the "foo" branch.

What branch am I on?

Two ways:

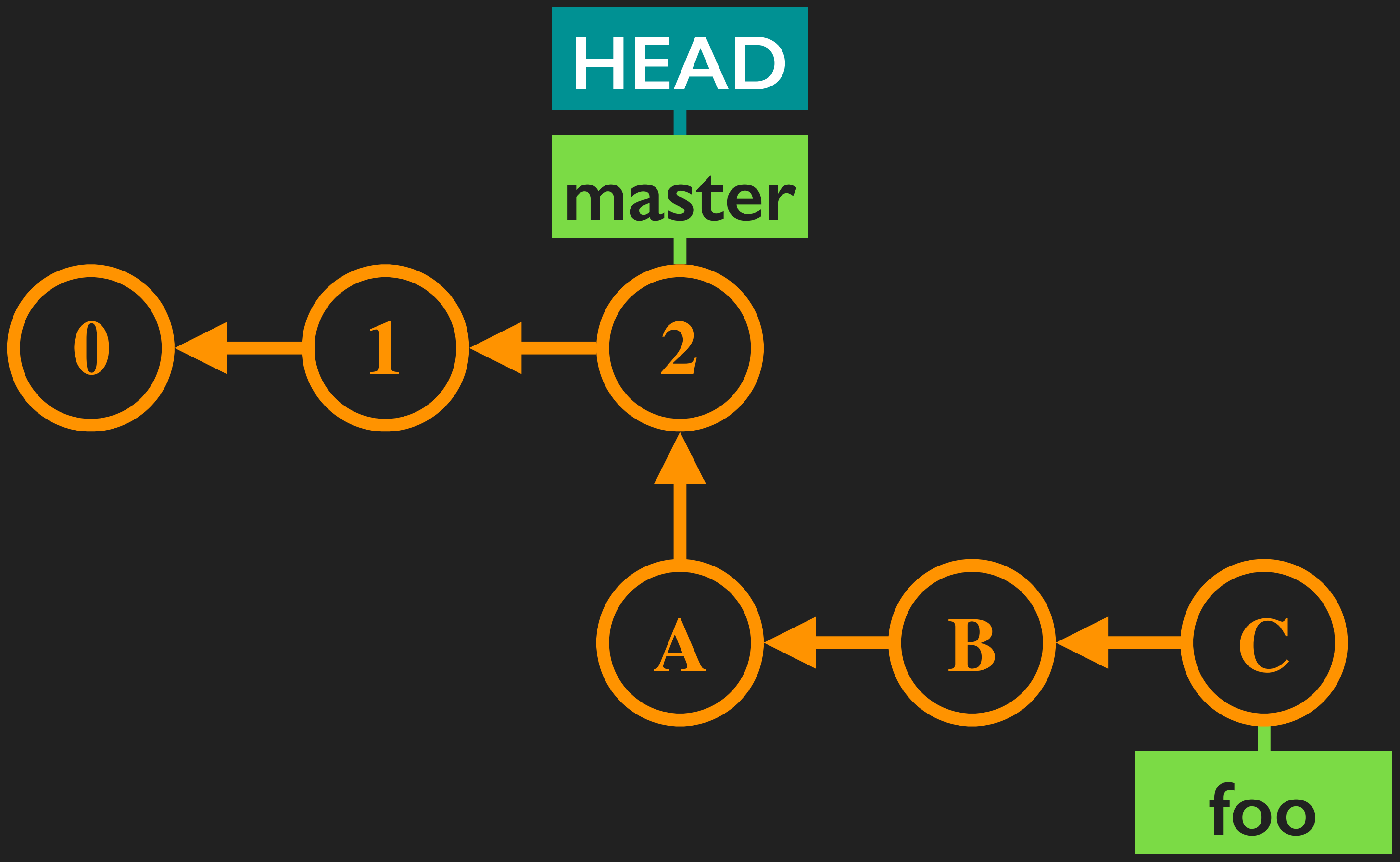
```
$ git status
On branch master
# ... rest of “git status output” ...
```

...or...

```
$ git branch
  foo
* master
  release
```

Merging branches

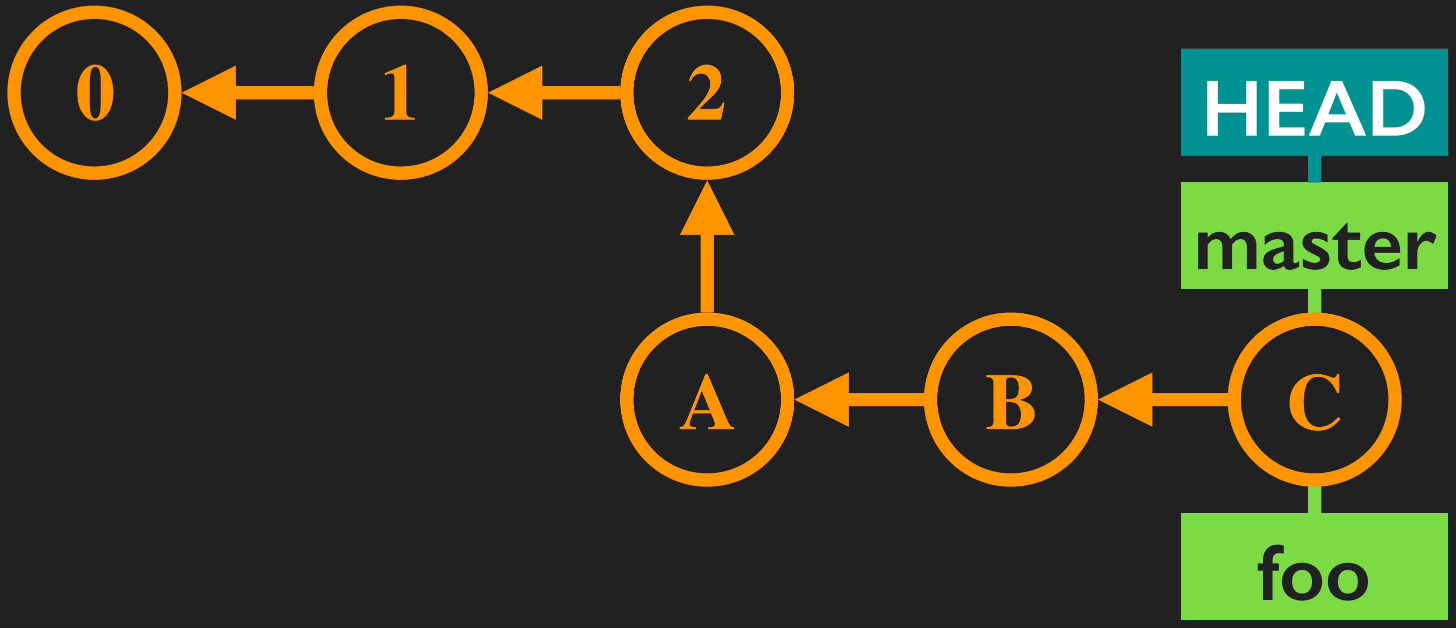
“`git merge`” integrates changes into the current branch.



Starting from the “master” branch…

Merging branches

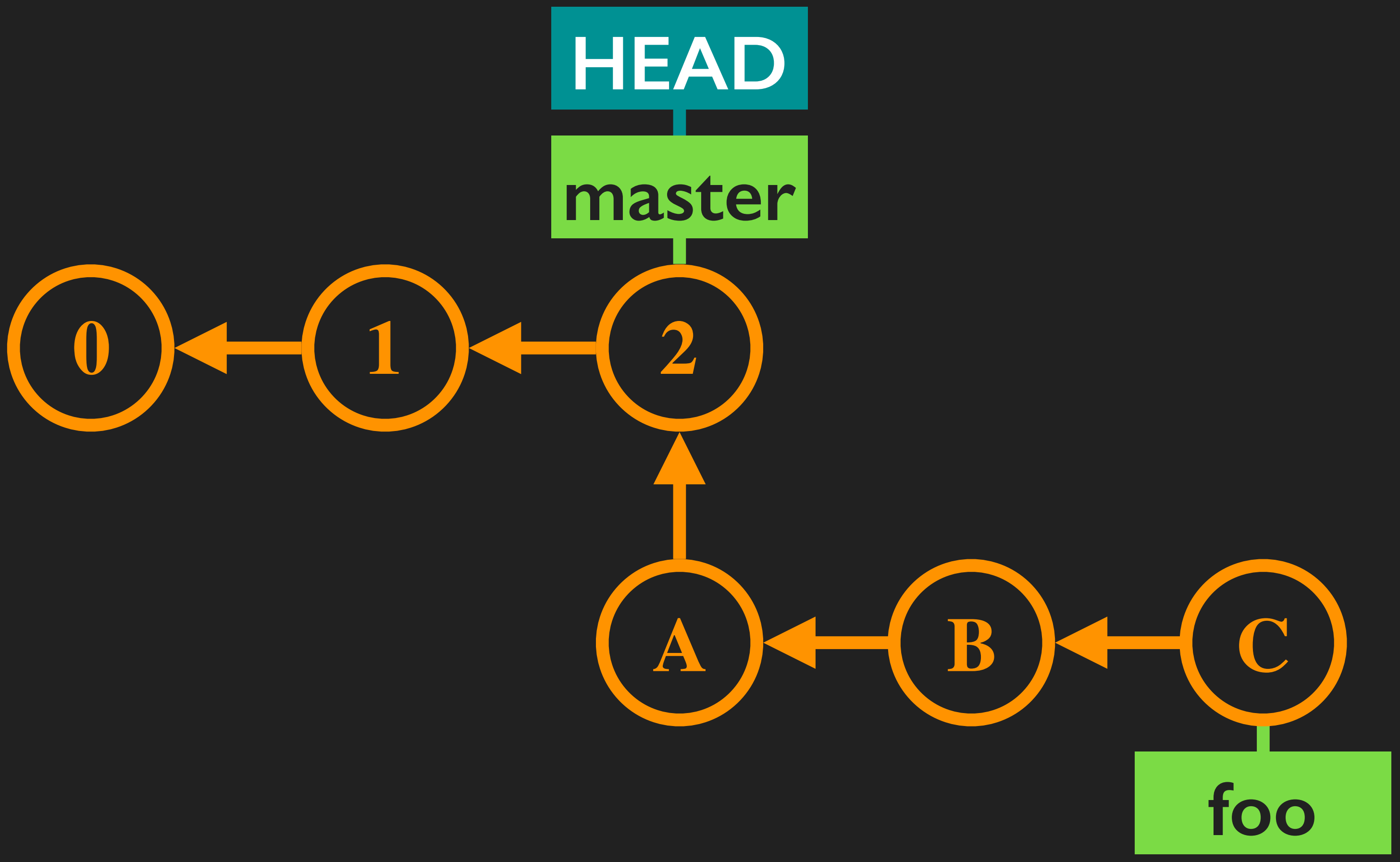
"`git merge`" integrates changes into the current branch.



…running "`git merge foo`" integrates changes from branch "foo" (commits A, B, and C) into our current branch ("master"). Has effect of advancing HEAD commit as well.

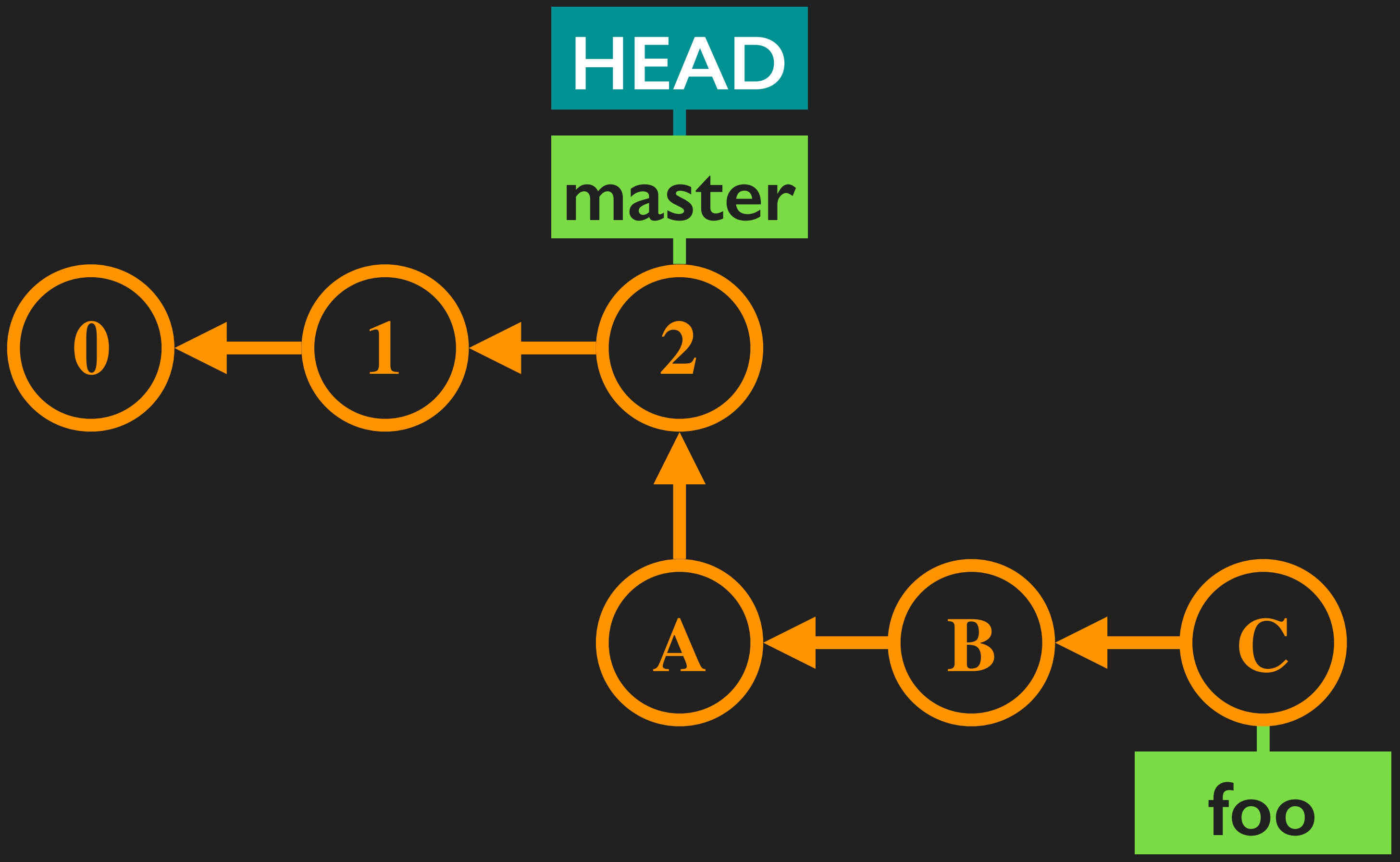
History need not be linear

We can commit to branches independent of each other.



History need not be linear

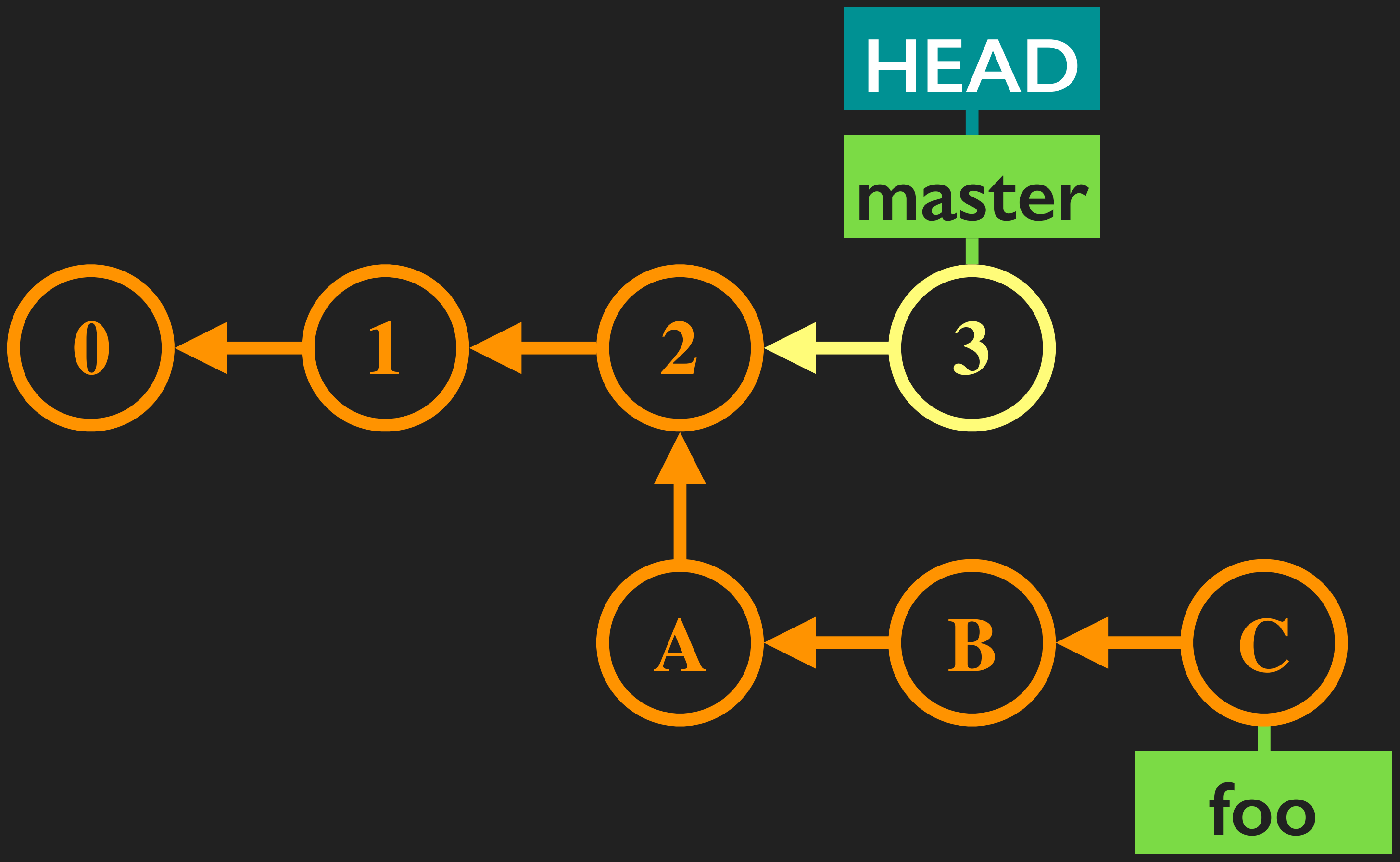
We can commit to branches independent of each other.



Starting from master and committing two times…

History need not be linear

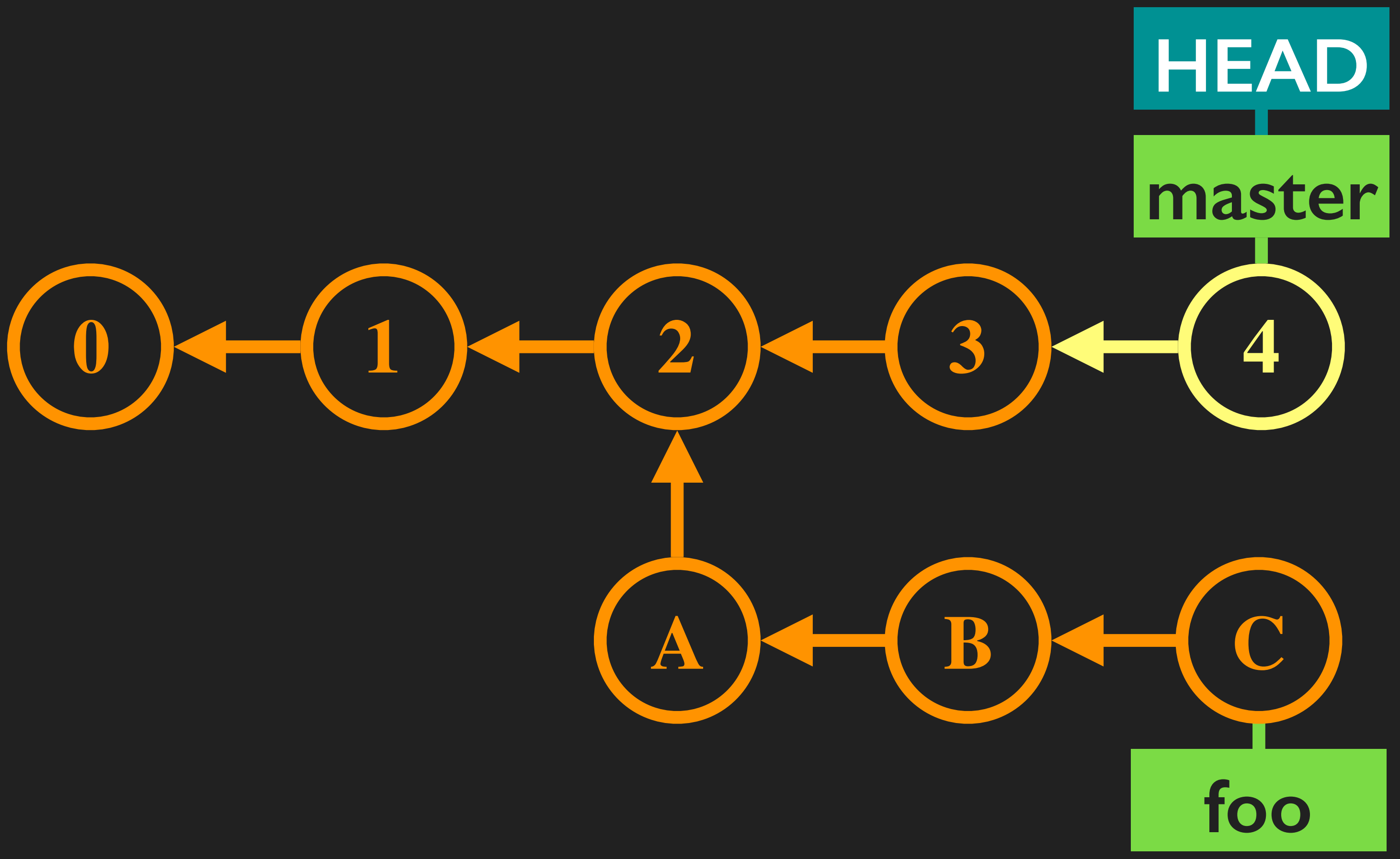
We can commit to branches independent of each other.



Starting from master and committing two times…

History need not be linear

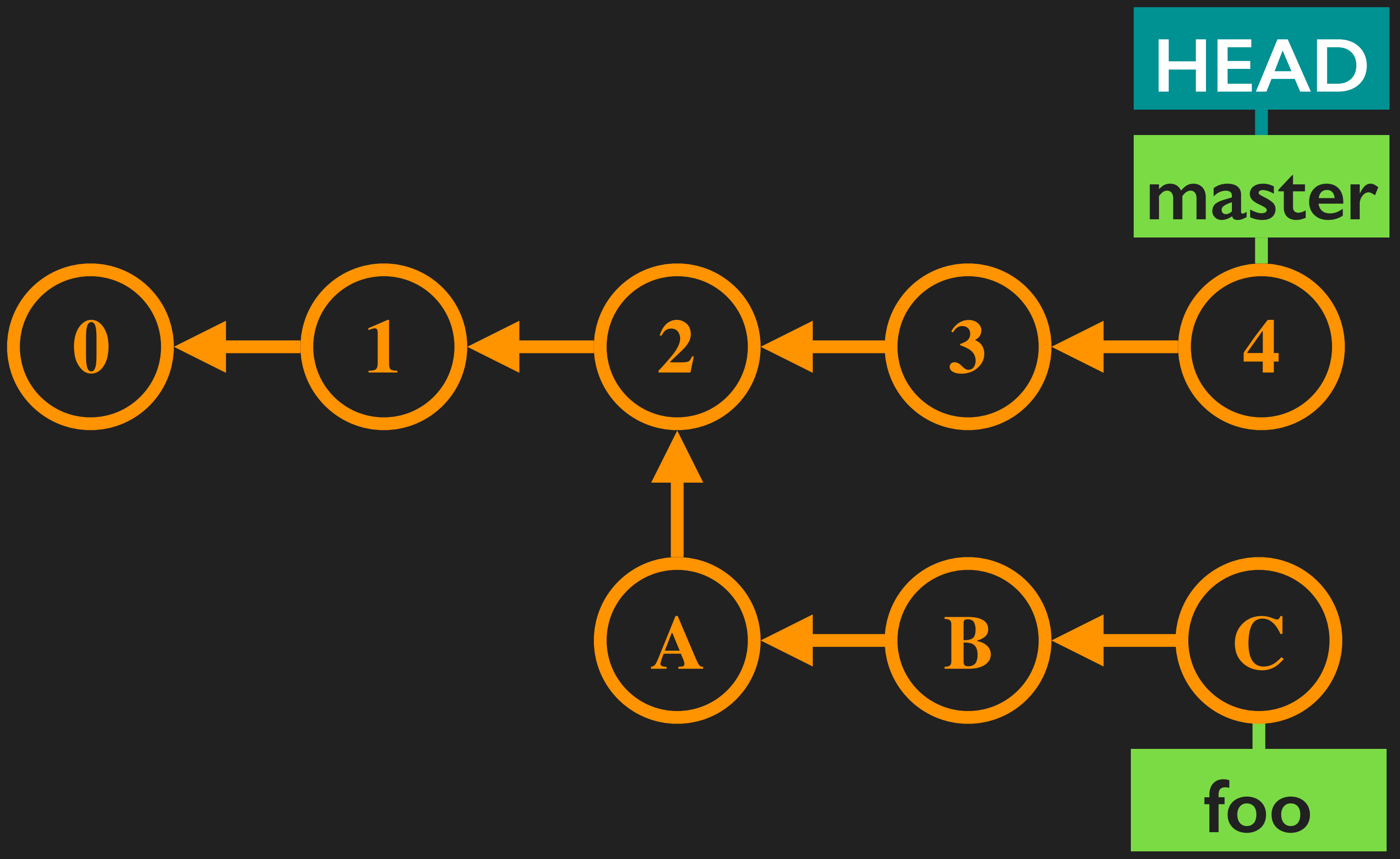
We can commit to branches independent of each other.



Starting from master and committing two times…

Merge commits

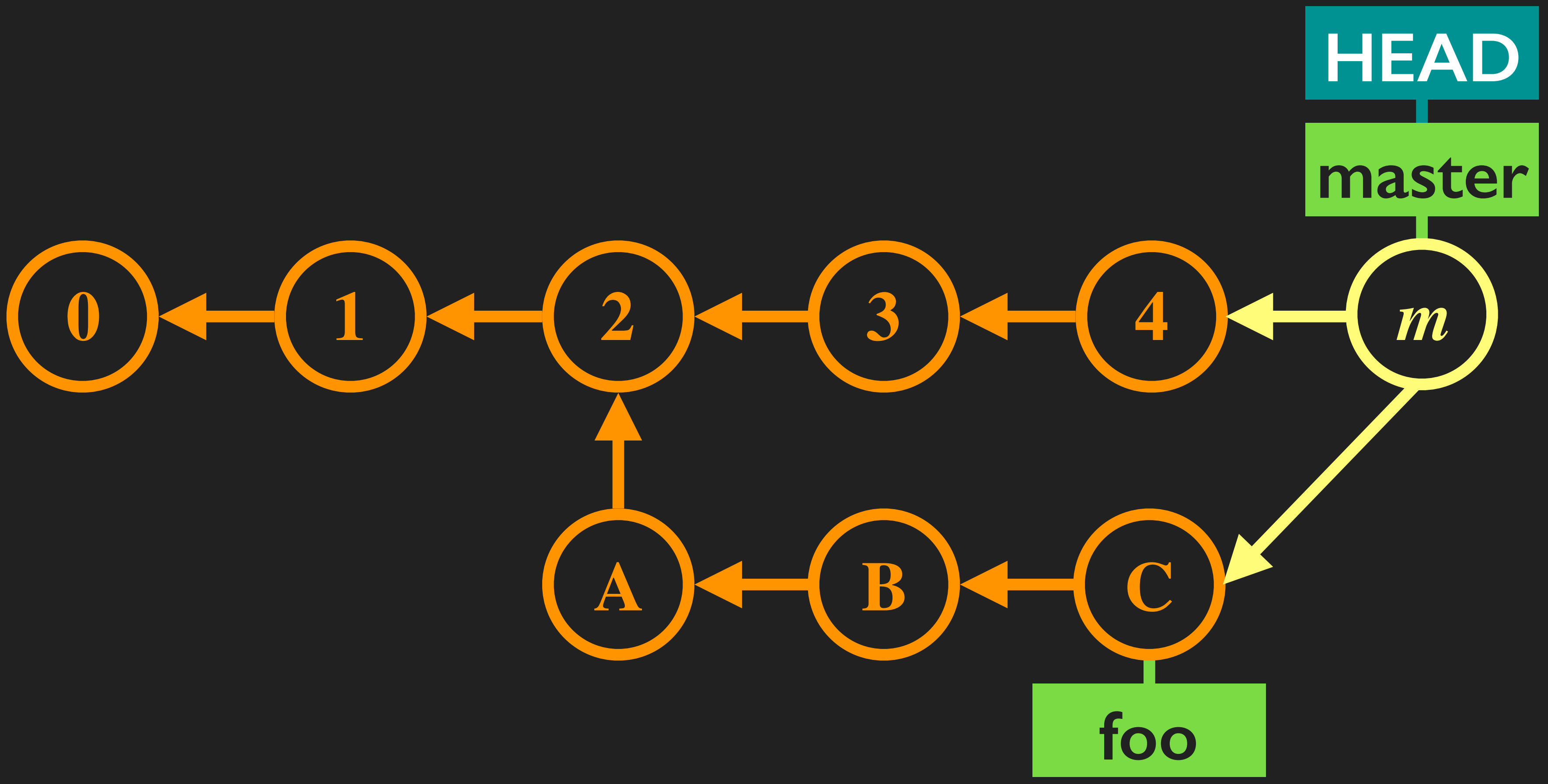
“merge” handles branches with divergent histories.



Starting from the “master” branch…

Merge commits

“merge” handles branches with divergent histories.



…running “git merge foo” agains integrates changes from branch “foo” into “master”. Commit *m* is known as a “**merge commit**”.

Merge conflicts

Git is usually smart enough to figure how to merge modifications, even if they’re in the same file.

Merge conflicts arise when git needs human intervention to figure out which modifications to files are “correct”.

Recognizing merge conflicts

Last message from “merge” command will let us know.

```
$ git merge foo
# ... other merge messages ...
CONFLICT (content): Merge conflict in my-code.py
# ... other merge messages ...
Automatic merge failed; fix conflicts and then commit the result.
$
```

Recognizing merge conflicts

“status” provides more details about merge conflicts.

```
  $ git status
  On branch master
  # ... other status messages ...
  You have unmerged paths.
    (fix conflicts and run “git commit”)
  # ... other status messages ...
  Unmerged paths:
    (use "git add/rm <file>..." as appropriate to mark
resolution)
    both modified:   my-code.py
  $
```

Recognizing merge conflicts

“status” provides more details about merge conflicts.

Conflict exists & what to do

```
$ git status
On branch master
# ... other status messages ...
You have unmerged paths.
  (fix conflicts and run “git commit”)
# ... other status messages ...
Unmerged paths:
  (use "git add/rm <file>..." as appropriate to mark
resolution)
  both modified:   my-code.py
$
```

Recognizing merge conflicts

“status” provides more details about merge conflicts.

```
$ git status
On branch master
# ... other status messages ...
You have unmerged paths.
  (fix conflicts and run “git commit”)
# ... other status messages ...
Unmerged paths:
  (use "git add/rm <file>..." as appropriate to mark 
resolution)
  both modified:   my-code.py
$
```

Which file(s) and the conflict type(s)

Resolving merge conflicts

Merge conflicts between “<<<<<<<” and “>>>>>>>”:

```
$ vi my-code.py
# ... other file contents ...
<<<<<<< HEAD
print “good morning, world!”
=======
print “good afternoon, world!”
>>>>>>> foo
# ... other file contents …
```

Resolving merge conflicts

Merge conflicts between “<<<<<<<” and “>>>>>>>”:

```
$ vi my-code.py
# ... other file conte
<<<<<<< HEAD
print “good morning, world!”
=======
print “good afternoon, world!”
>>>>>>> foo
# ... other file contents …
```

code from our branch (“master”)

Resolving merge conflicts

Merge conflicts between “<<<<<<<” and “>>>>>>>”:

```
$ vi my-code.py
# ... other file contents ...
<<<<<<< HEAD
print “good morning, world!”
=======
print “good afternoon, world!”
>>>>>>> foo
# ... other file conte
```

code from other branch (“foo”)

Resolving merge conflicts

Up to you to decide what the correct code is

```
# while editing “my-code.py” ...
# ... other file contents ...
print “good afternoon, world!”
# ... other file contents ...
```

Be aware: there may be >1 merge conflict per file!

Resolving merge conflicts

Complete the merge via normal commit process

```
# Save “my-code.py” and quit
$ git add my-code.py
```

Resolving merge conflicts

Complete the merge via normal commit process

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
        # ... other modified/added/deleted files ...
        Modified:   my-code.py
        # ... other modified/added/deleted files ...
```

Resolving merge conflicts

Complete the merge via normal commit process

```
$ git commit
# Editor will appear for you to provide a commit message.
# Default “Merge branch ‘foo’ into master” usually ok.
[master 7d1bc7e] Merge branch 'foo' into master
```

Remotes

Can work collaboratively with others using **remotes**, which are “copies” of our repository in other places.

Why “copies” in quotes? Key feature of distributed VCS: not all commits need be shared among repositories.

Remotes are located via URLs (https, ssh, git, file, etc.) but are referred to using (local) names.

Remotes

“git remote” lists the (local) names of known remotes.

```
$ git remote
origin
upstream
```

Remotes

“git remote -v” to get the URLs for our remotes.

```
  $ git remote -v
  origin git@github.com:chenghlee/UTbiocomputing2015.git (fetch)
  origin git@github.com:chenghlee/UTbiocomputing2015.git (push)
  upstream https://github.com/sjspielman/UTbiocomputing2015.git
(fetch)
  upstream https://github.com/sjspielman/UTbiocomputing2015.git
(push)
```

The “origin” remote

*When we clone a repository, git automatically creates default “**origin**” remote for that source.*

“origin” usually is a central server (e.g., GitHub) where we can share code with other developers/users.

Remote branches

Remotes, like any repository, have one or more branches (usually at least a “master” branch).

Remote branches have a “/” in their name separating the remote and branch name. E.g., the “master” branch on “origin” is called “**origin/master**”.

Local repository can interact with remotes by either:

- Getting commits from remote branches (**fetch**/**pull**)
- Sending commits to remotes branches (**push**)

Tracking branches

Local & remote branches need not be related/interact.

Tracking branch: a local branch configured with a direct relationship to a remote branch. Useful because it helps define defaults when we fetch, push, and pull.

When you clone a repository, git automatically sets the local "master" as a tracking branch of "origin/master".

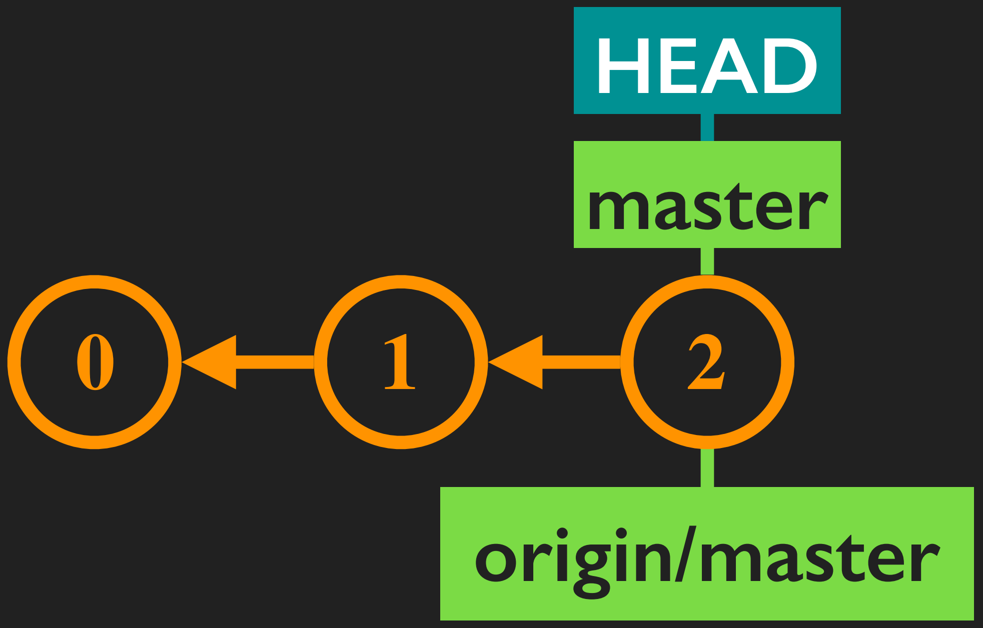
“fetch” gets updates from remotes

```
$ git fetch origin
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 9 (delta 4), reused 0 (delta 0)
Unpacking objects: 100% (9/9), done.
From bitbucket.org:lab7io/biobuilds
   dc7ea0d..e0ab75f  master     -> origin/master
   97ed4ee..b5fb03e  release    -> origin/release
 * [new branch]      rnastar    -> origin/rnastar
```

“git fetch origin” updates the local repository with information about *all* branches of the “origin” remote.

Using remote branches

Think of remote branches as “read-only” local branches.

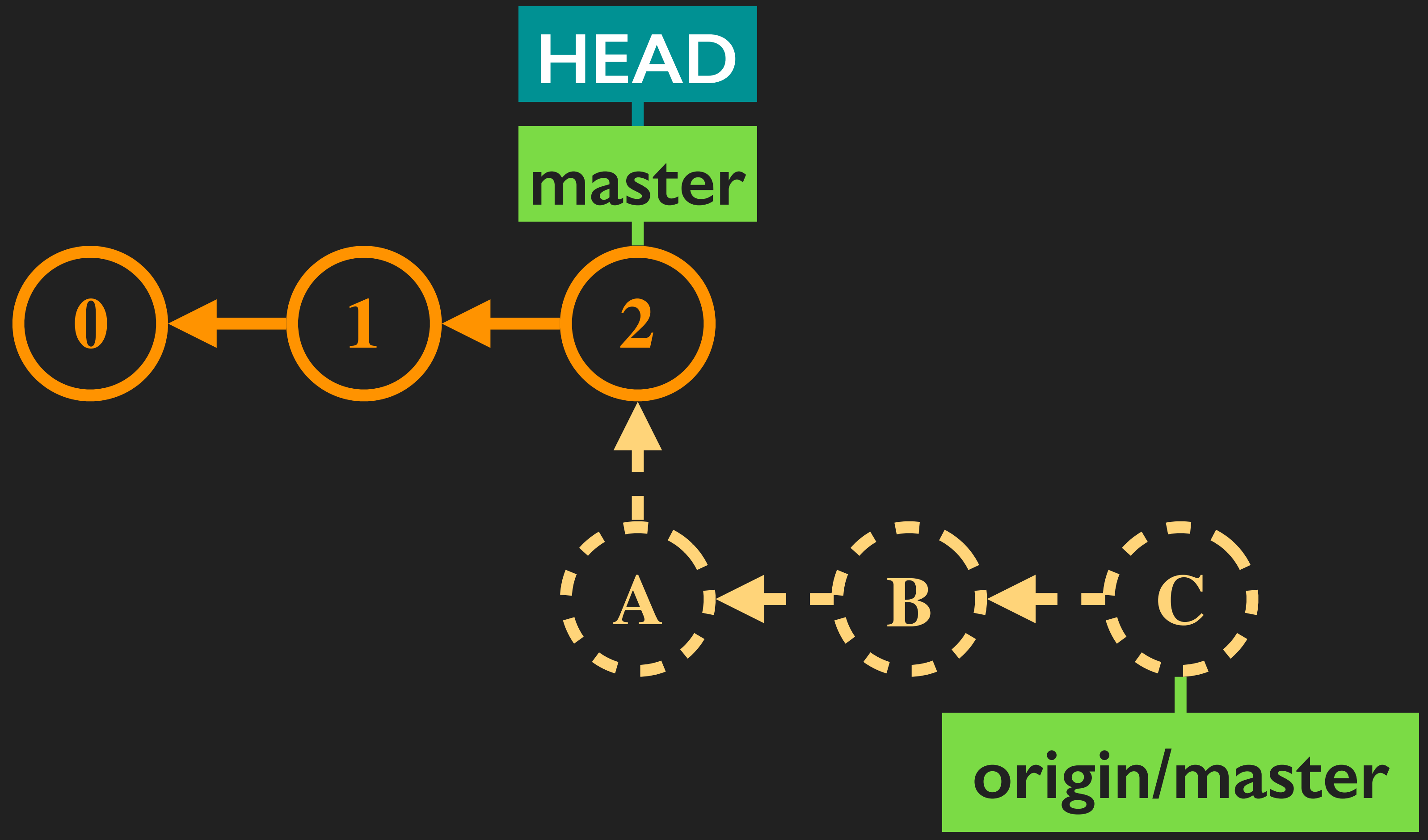


You can’t* commit directly to it, but you can do other things like merge it with your local branch.

* “can’t” == “shouldn’t”

Using remote branches

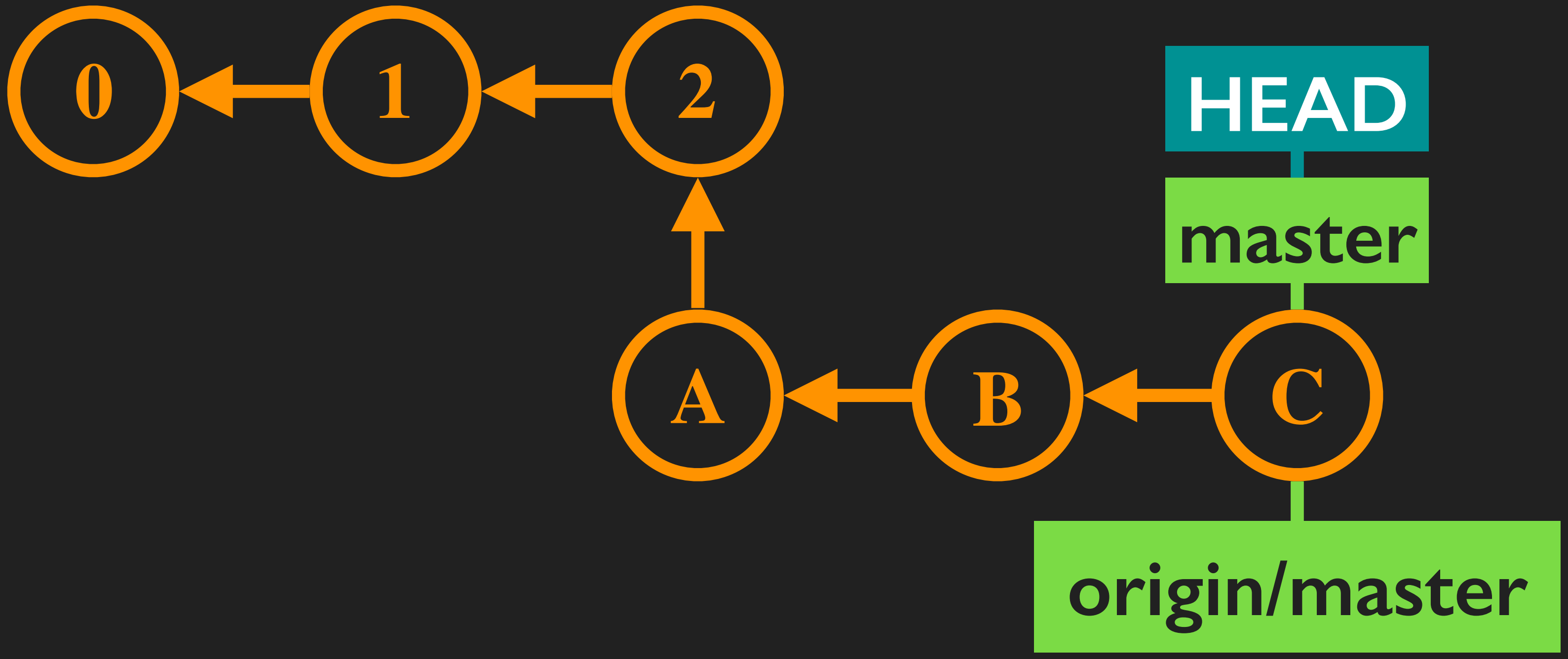
“fetch” gets commits & updates remote branch labels.



After “`git fetch origin`”…

Using remote branches

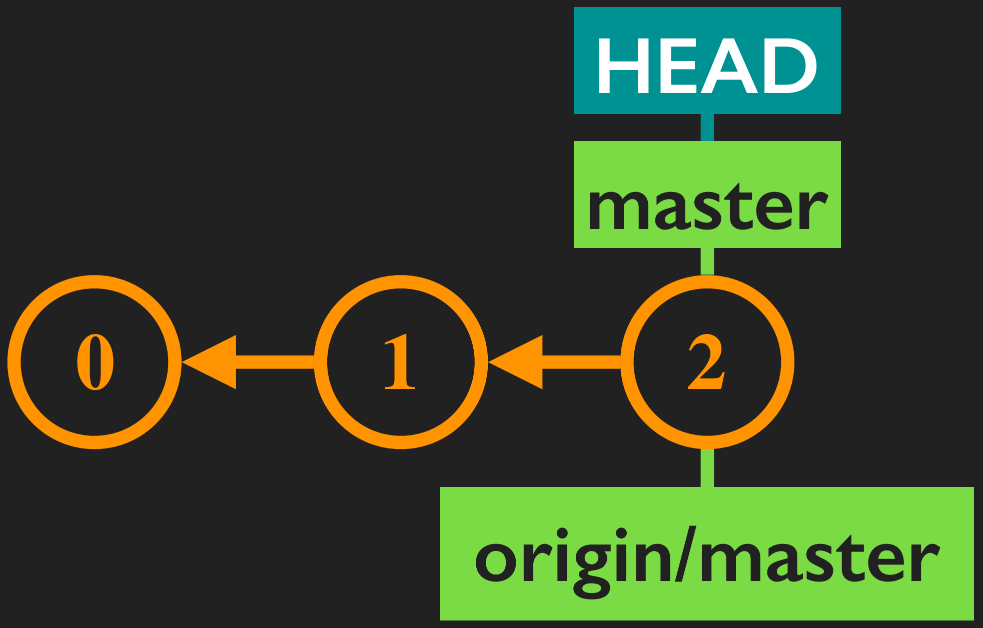
“merge” integrates remote code changes with yours.



After “`git merge origin/master`”…

Remotes branches key to collaboration

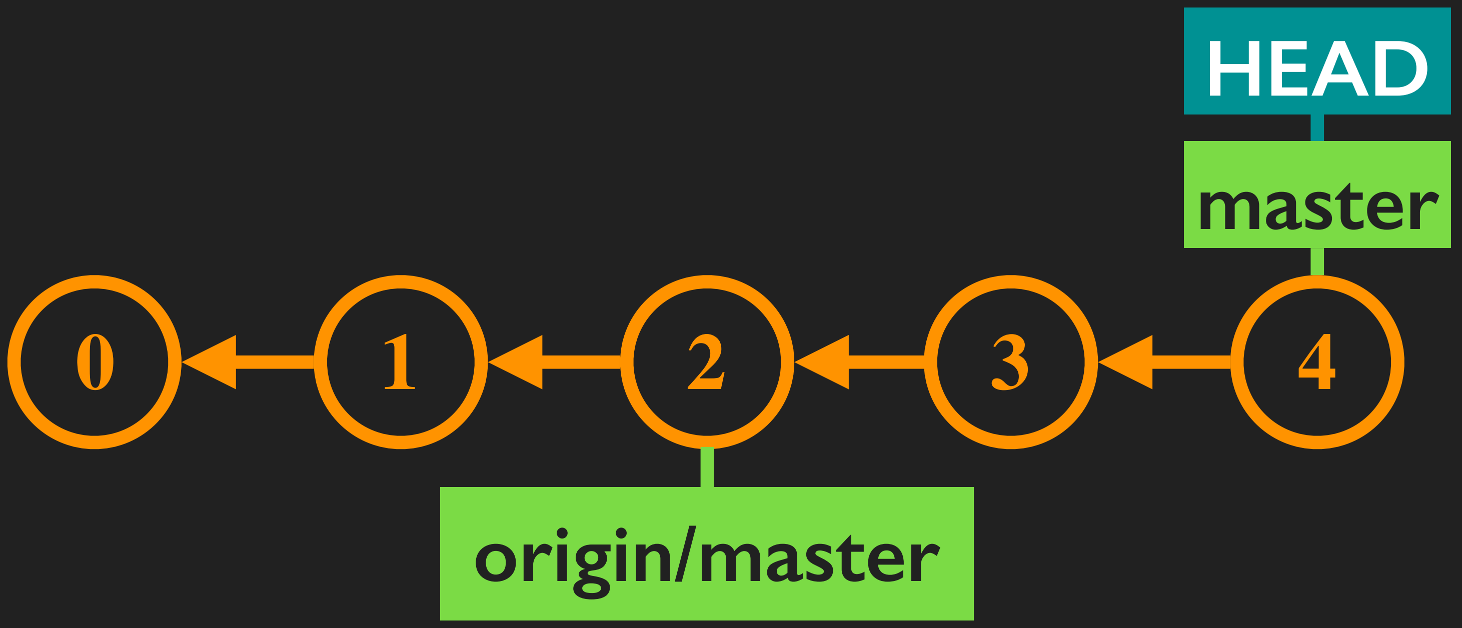
Remotes allow you to work independently.



The situation after you’ve just cloned a repository

Remotes branches key to collaboration

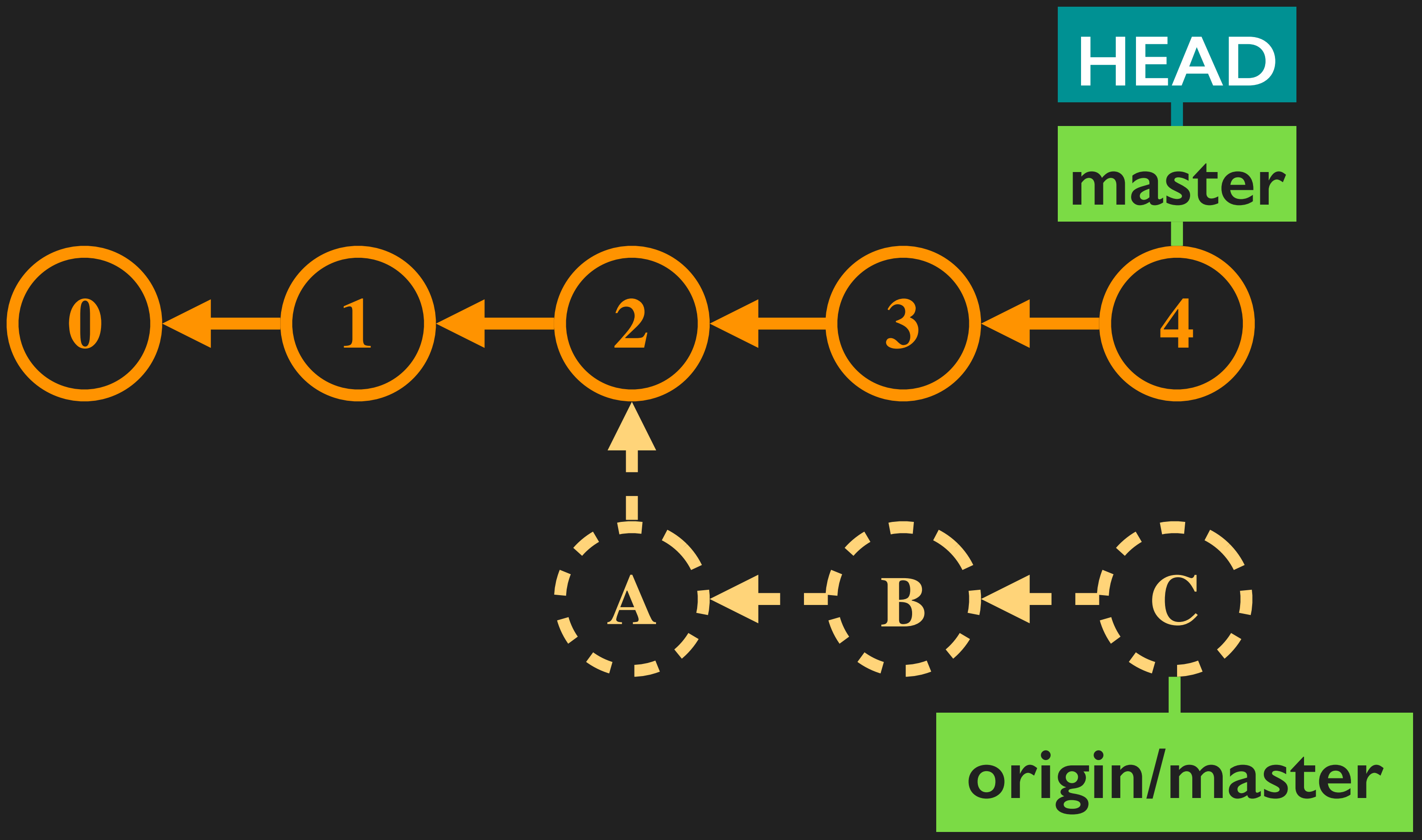
Commits only affect your local branch.



“`git commit`” advances “master” but not “origin/master”.

Remotes branches key to collaboration

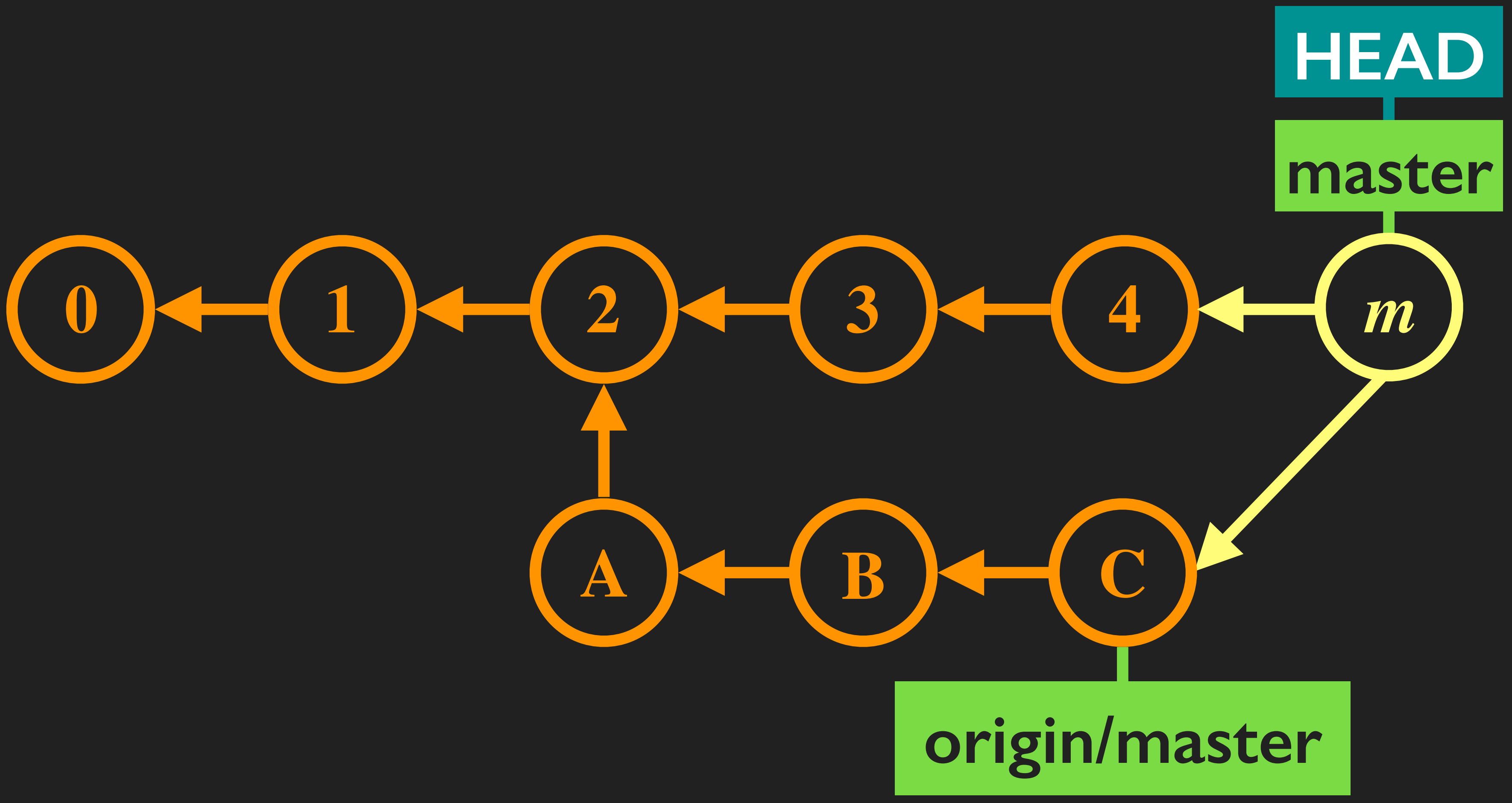
Again, use “fetch” to get other people’s contributions…



“`git fetch origin`”…

Remotes branches key to collaboration

…and “merge” to integrate their code with yours.



“`git merge origin/master`”…

“pull” as a short-cut

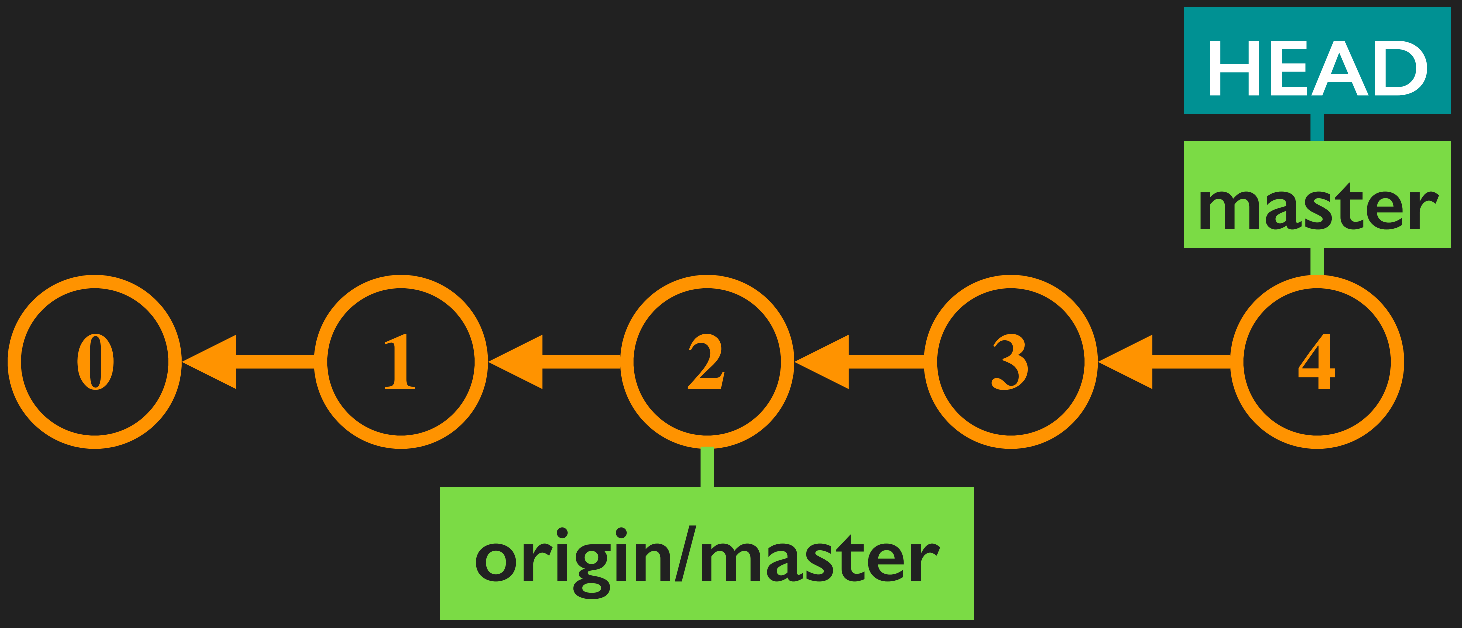
The fetch-then-merge pattern is *really* common.

If your local branch is also a tracking branch, you can use “`git pull`” to fetch and merge with a single command.

Works because tracking branches know which remote and remote branch to use.

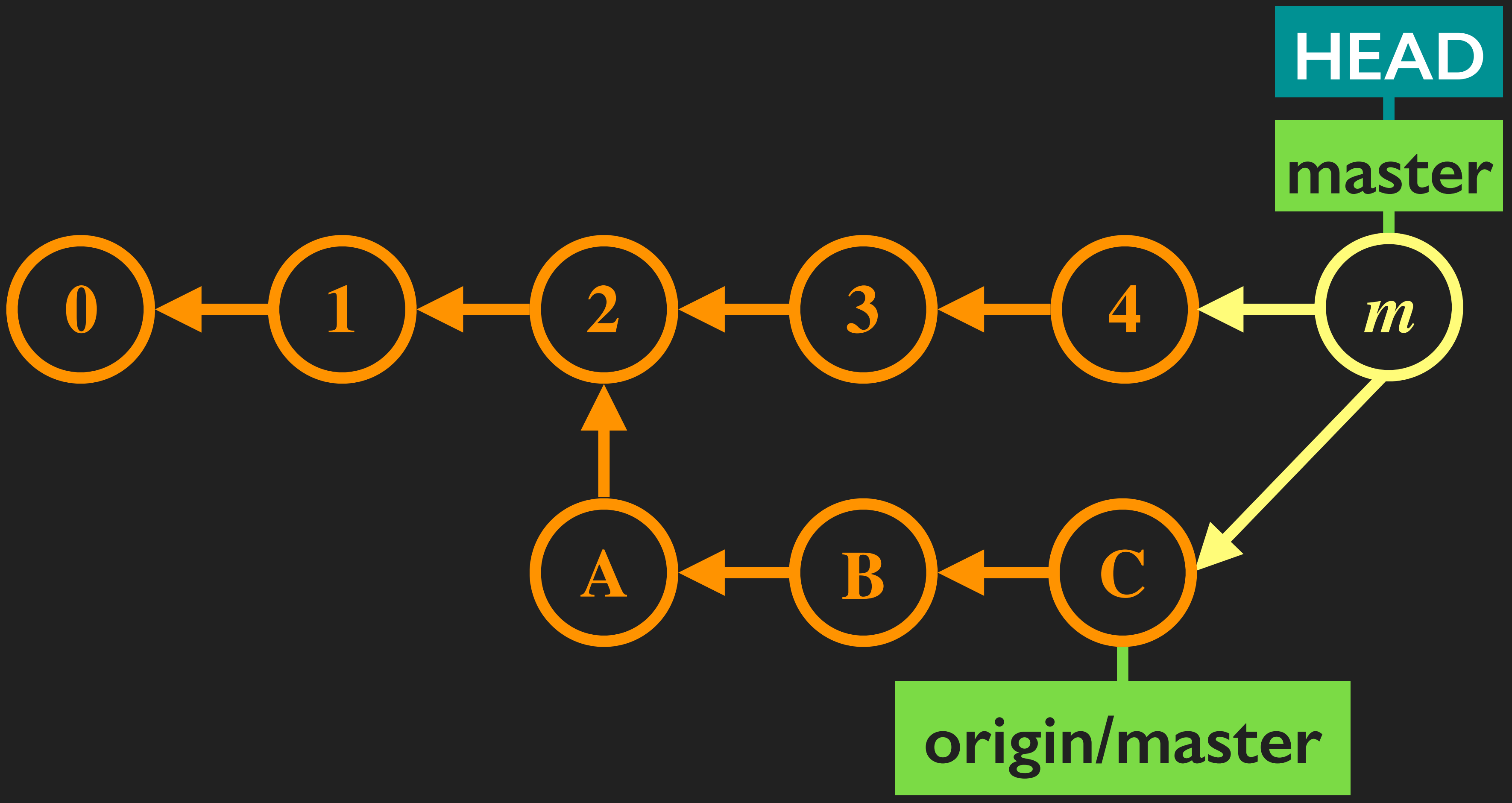
“pull” as a short-cut

Assuming “master” is tracking “origin/master”...



“pull” as a short-cut

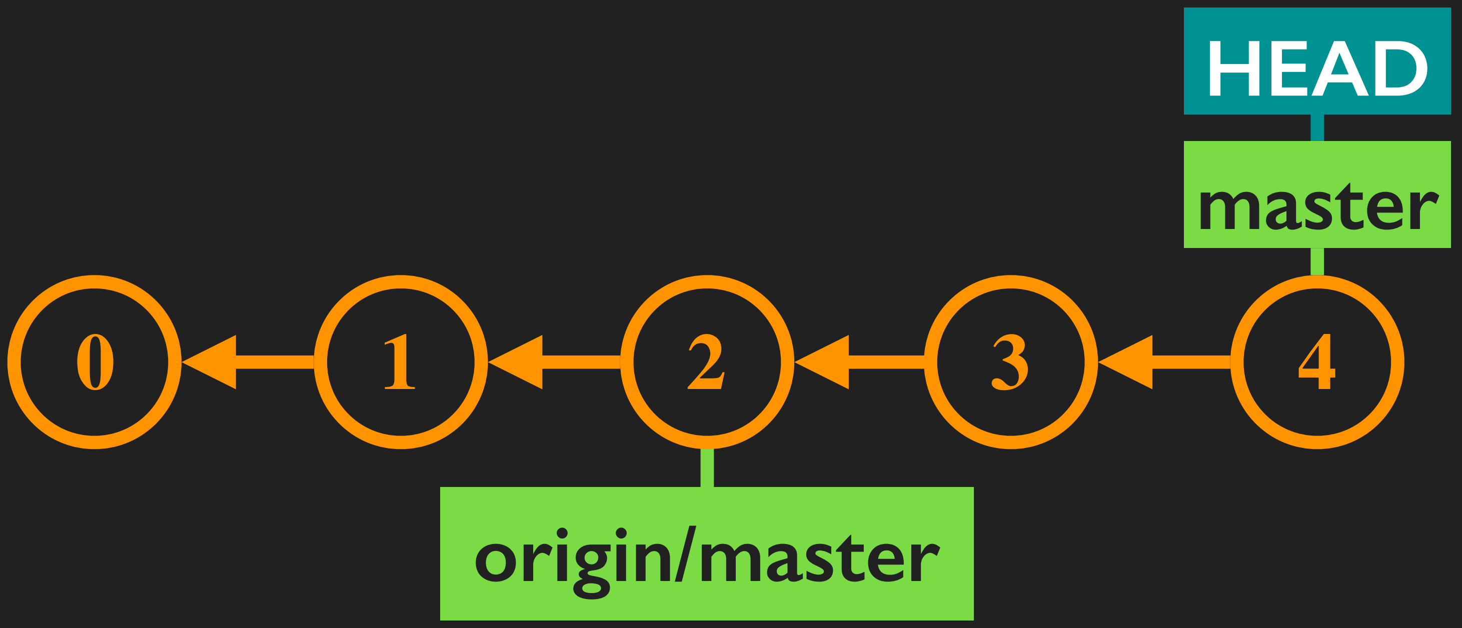
…“`git pull`” merges in changes with a single command.



Essentially performs a “`git fetch origin master`” followed by a “`git merge origin/master`”.

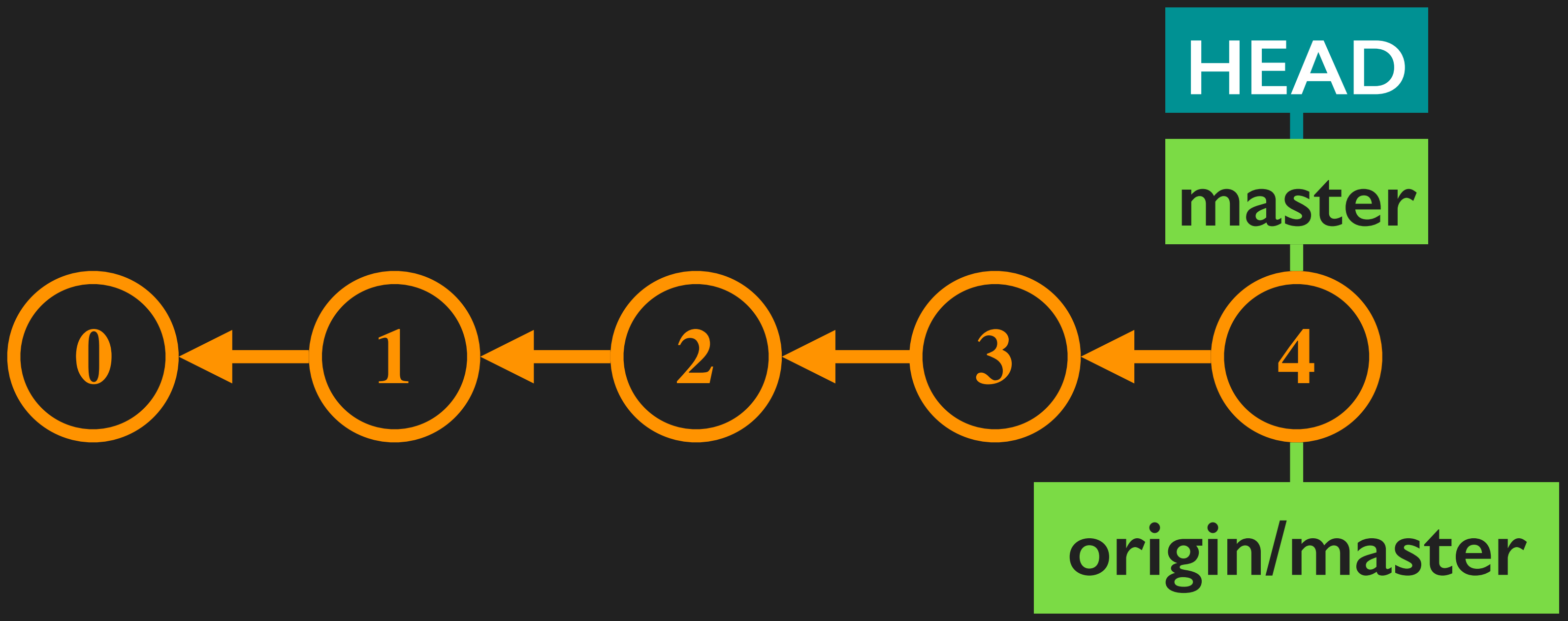
“push” lets us share code

Say we’ve made changes we’d like to share…



“push” lets us share code

…a “`git push origin master`” makes that code public.



Commits 3 & 4 are now shared and available to anyone who runs “`git fetch origin`”.

“push” has version-specific quirks

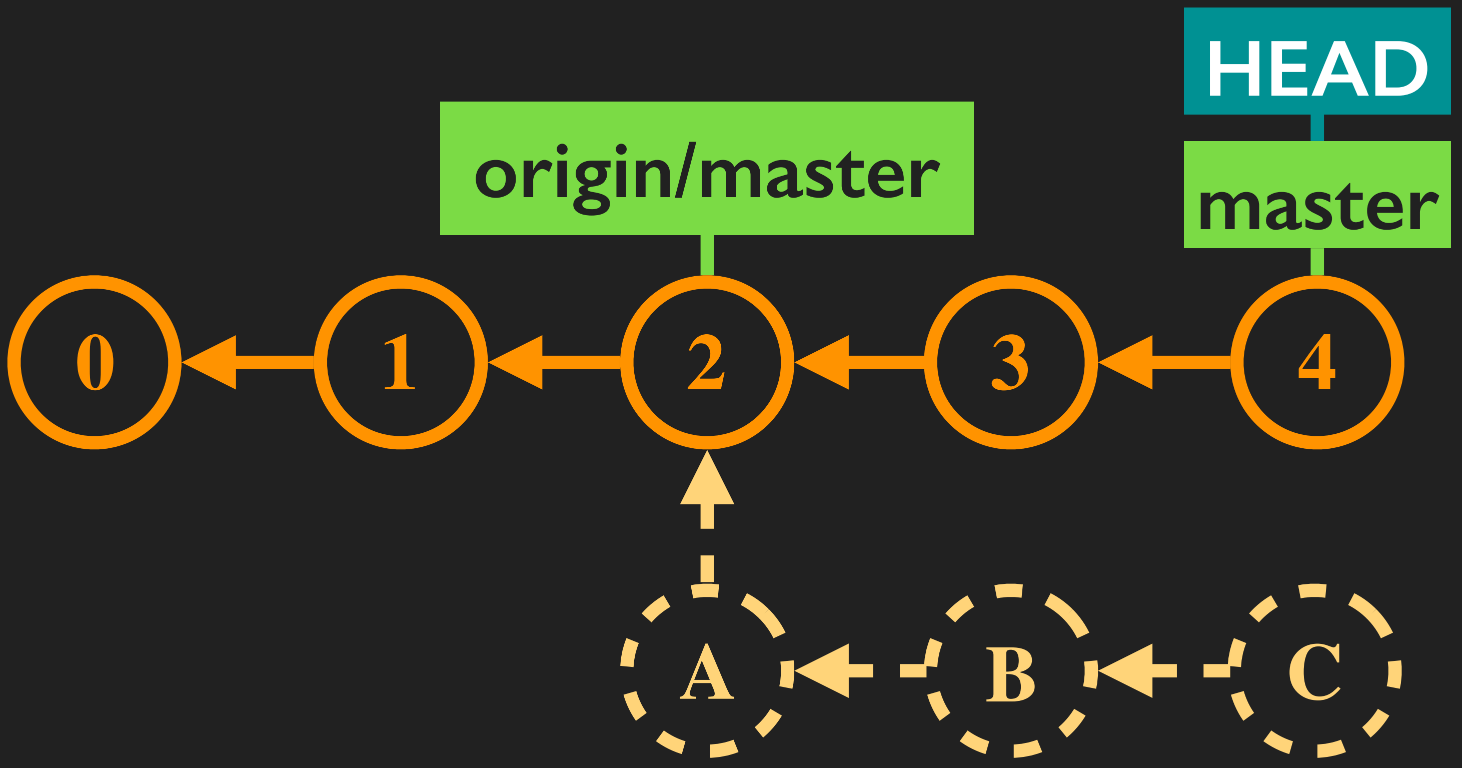
TL;DR: Always be explicit about the remote server & branch you’re pushing to (i.e., “`git push origin master`”, not “`git push`”), even with tracking branches.

In older versions of git, default behavior of a plain “`git push`” is to push *all** local branches to “origin”, instead of just the (tracking) branch you’re sitting on.

In git ≥ 1.8, the default behavior could be tweaked by a config option. In git ≥ 2.0, the default behavior switched to be more-intuitive pushing of just your tracking branch.

Best practice: “pull” before “push”

By default, “push” will refuse to destroy existing history.

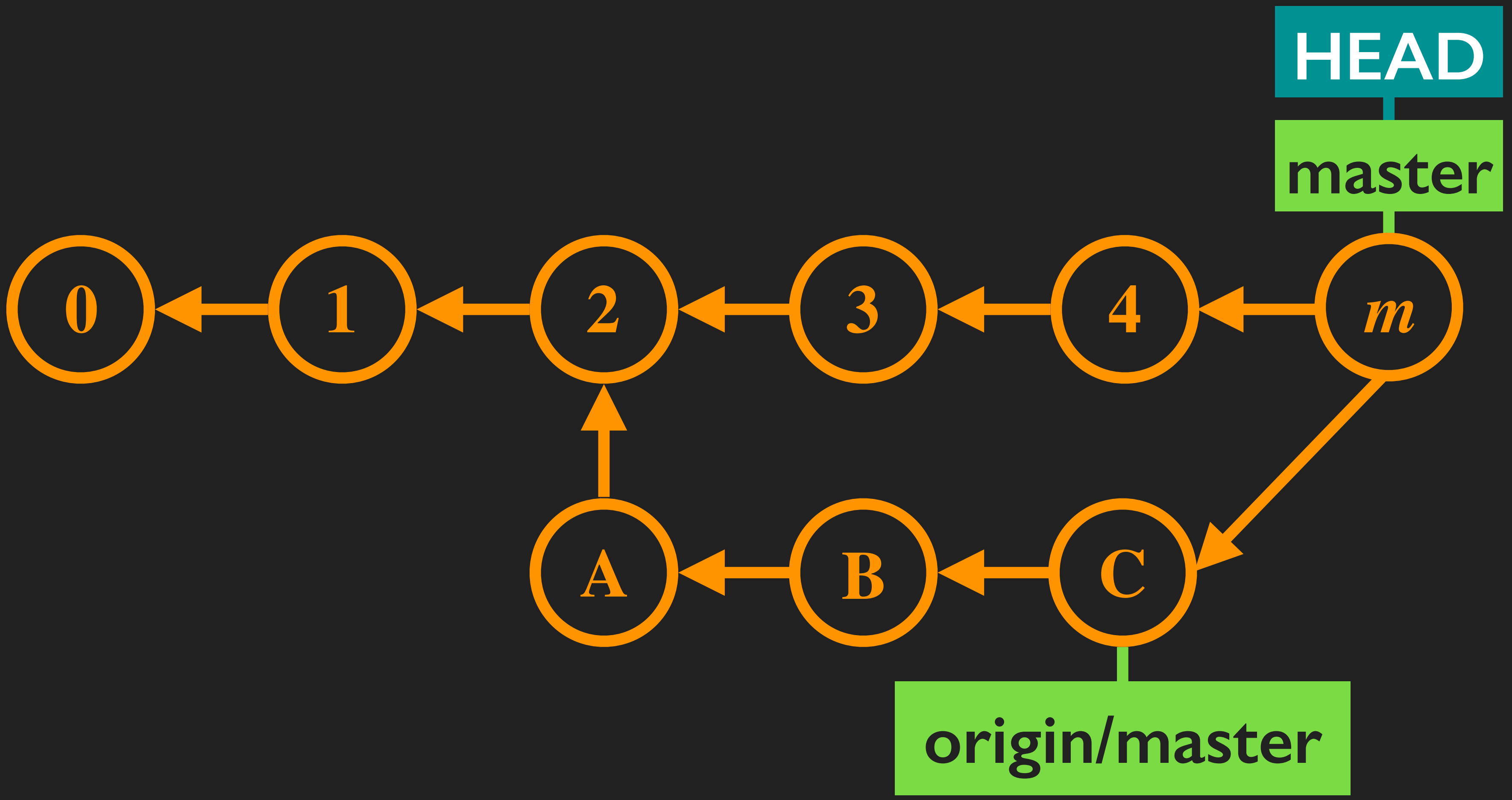


A, B, and C are commits added since your last “git fetch origin”.

In this case, “`git push origin master`” will complain and fail.

Best practice: “pull” before “push”

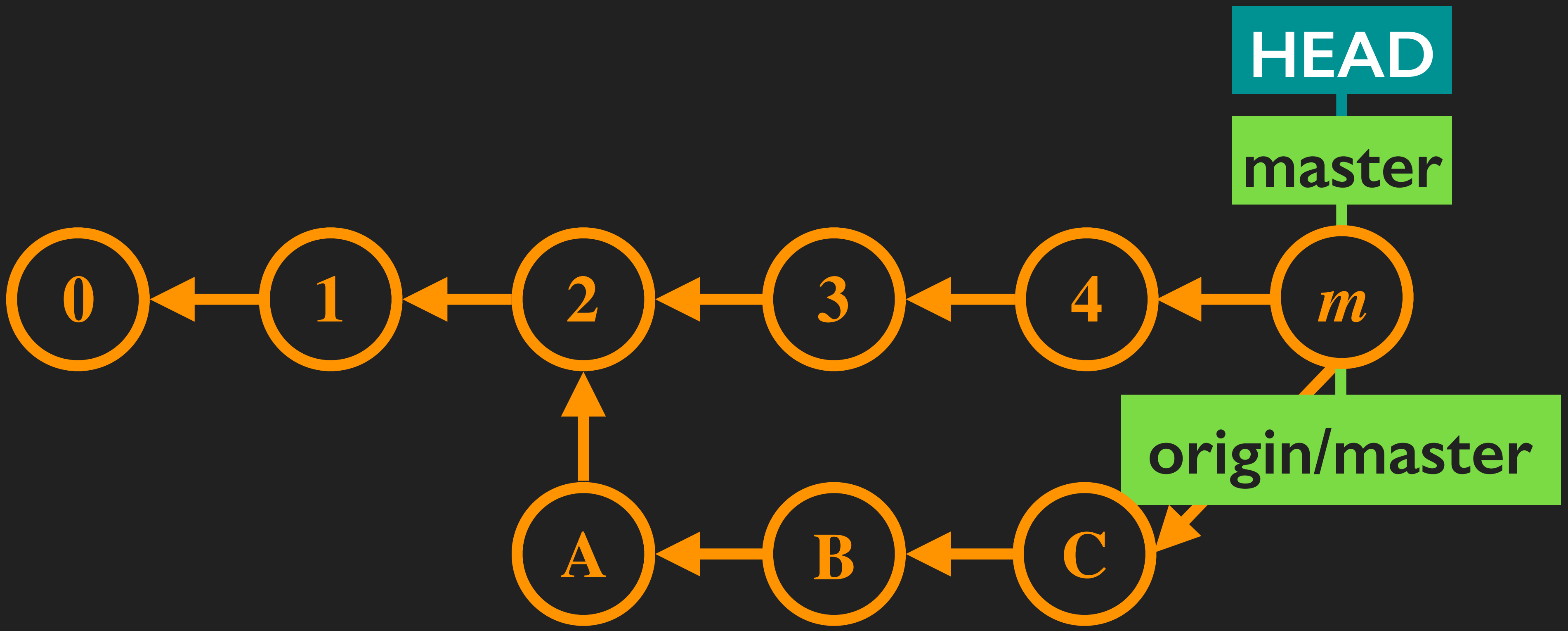
To avoid errors, run a “git pull” before you “push”.



A “`git pull`” ensures your repository’s world view matches that of “origin” (i.e., commits A, B, and C exist in both repositories).

Best practice: “pull” before “push”

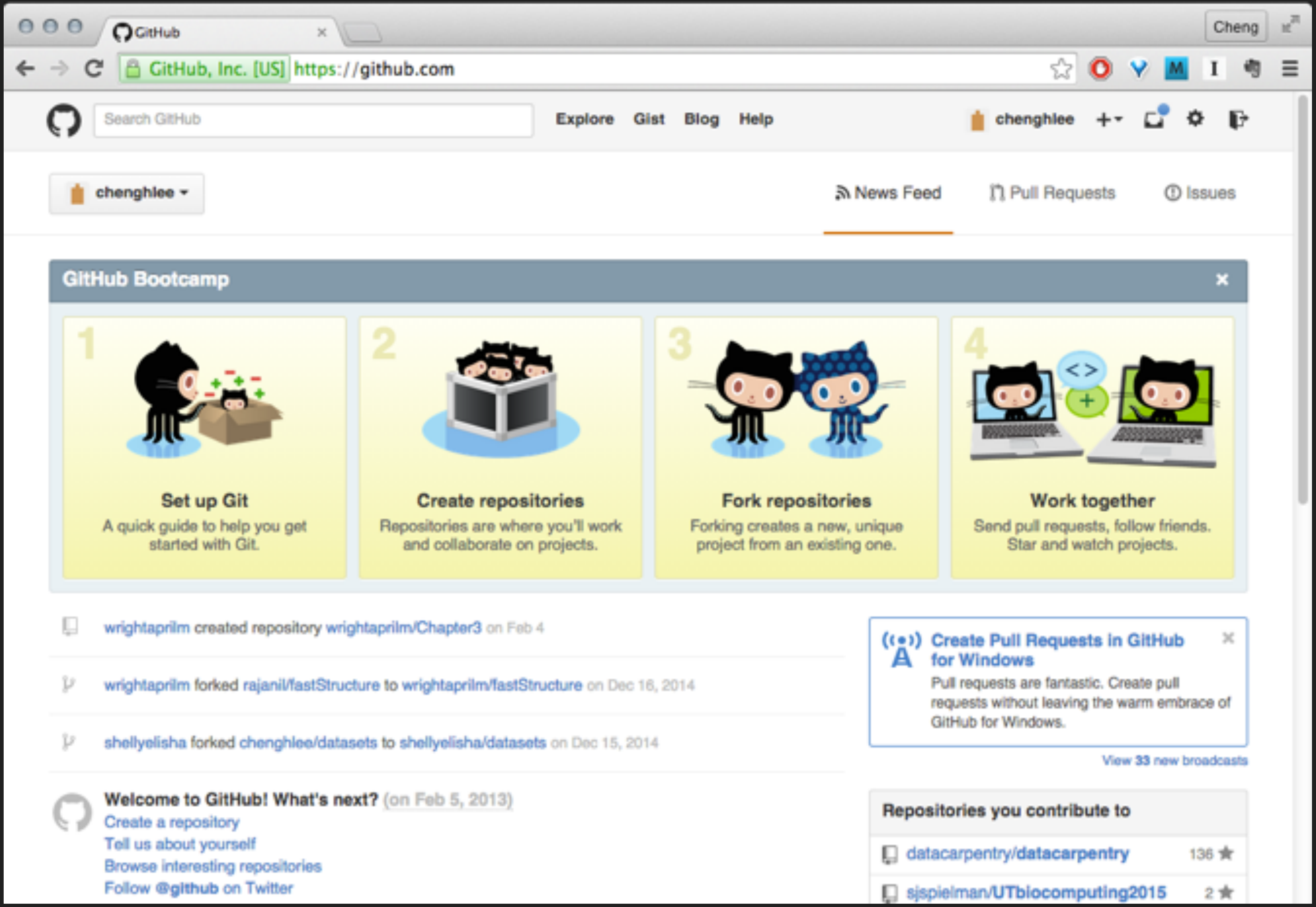
To avoid errors, run a “git pull” before you “push”.



“`git push origin master`” works once commit histories match.

Github

Popular site for hosting git repositories.



Set up Git
A quick guide to help you get started with Git.

Create repositories
Repositories are where you'll work and collaborate on projects.

Fork repositories
Forking creates a new, unique project from an existing one.

Work together
Send pull requests, follow friends. Star and watch projects.

wrightaprilm created repository wrightaprilm/Chapter3 on Feb 4

wrightaprilm forked rajanil/fastStructure to wrightaprilm/fastStructure on Dec 16, 2014

shellyelisha forked chenghlee/datasets to shellyelisha/datasets on Dec 15, 2014

Welcome to GitHub! What's next? (on Feb 5, 2013)
Create a repository
Tell us about yourself
Browse interesting repositories
Follow @github on Twitter

Create Pull Requests in GitHub for Windows
Pull requests are fantastic. Create pull requests without leaving the warm embrace of GitHub for Windows.

View 33 new broadcasts

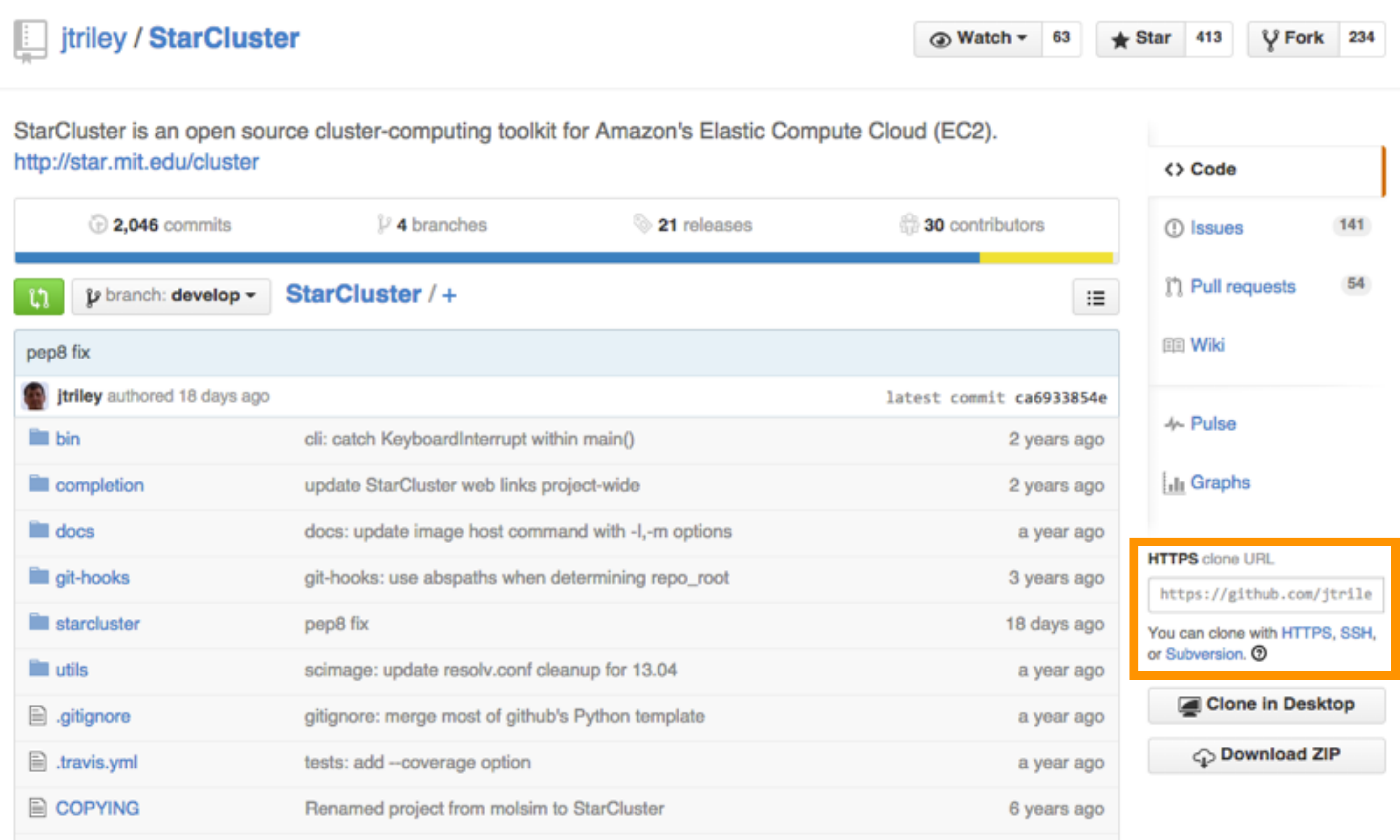
Repositories you contribute to

datacarpentry/datacarpentry	136 ★
sjspielman/UTbiocomputing2015	2 ★

Important to remember: GitHub != git

Using Github as “origin”

Generally can’t edit directly in Github: clone repo. to your computer using URL provided in the sidebar to work.



StarCluster is an open source cluster-computing toolkit for Amazon's Elastic Compute Cloud (EC2). http://star.mit.edu/cluster

2,046 commits · 4 branches · 21 releases · 30 contributors

branch: develop · StarCluster / +

pep8 fix

jtriley authored 18 days ago · latest commit ca6933854e

bin	cli: catch KeyboardInterrupt within main()	2 years ago
completion	update StarCluster web links project-wide	2 years ago
docs	docs: update image host command with -I,-m options	a year ago
git-hooks	git-hooks: use abspaths when determining repo_root	3 years ago
starcluster	pep8 fix	18 days ago
utils	scimage: update resolv.conf cleanup for 13.04	a year ago
.gitignore	gitignore: merge most of github's Python template	a year ago
.travis.yml	tests: add --coverage option	a year ago
COPYING	Renamed project from molsim to StarCluster	6 years ago

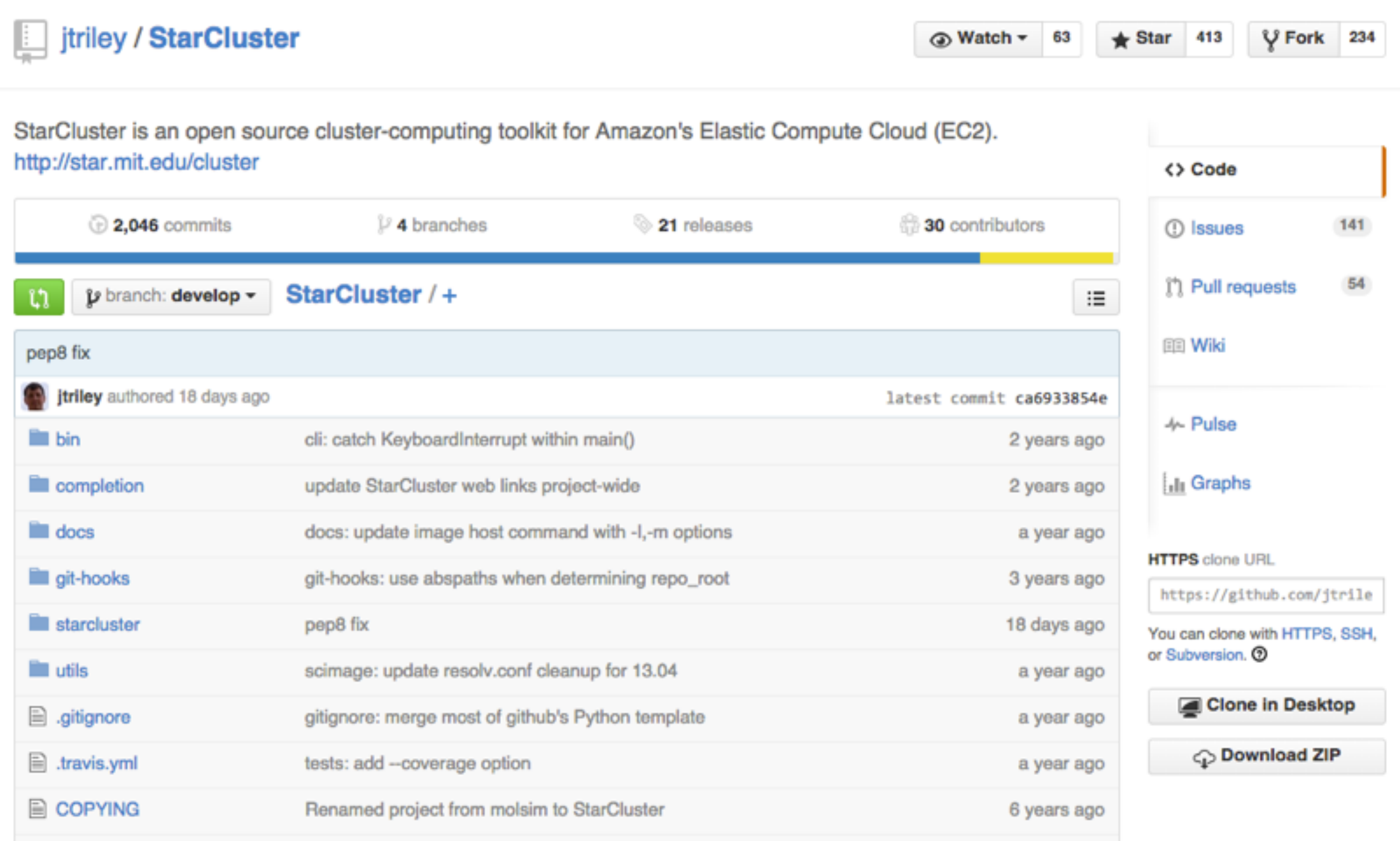
HTTPS clone URL

`https://github.com/jtrile`

You can clone with HTTPS, SSH, or Subversion.

Using Github as “origin”

Cloning sets Github as the “origin” remote. (“Nothing special” about GitHub; it acts like “origin” should.)



StarCluster is an open source cluster-computing toolkit for Amazon's Elastic Compute Cloud (EC2).
http://star.mit.edu/cluster

pep8 fix

jtriley authored 18 days ago latest commit ca6933854e

bin	cli: catch KeyboardInterrupt within main()	2 years ago
completion	update StarCluster web links project-wide	2 years ago
docs	docs: update image host command with -I,-m options	a year ago
git-hooks	git-hooks: use abspaths when determining repo_root	3 years ago
starcluster	pep8 fix	18 days ago
utils	scimage: update resolv.conf cleanup for 13.04	a year ago
.gitignore	gitignore: merge most of github's Python template	a year ago
.travis.yml	tests: add --coverage option	a year ago
COPYING	Renamed project from molsim to StarCluster	6 years ago

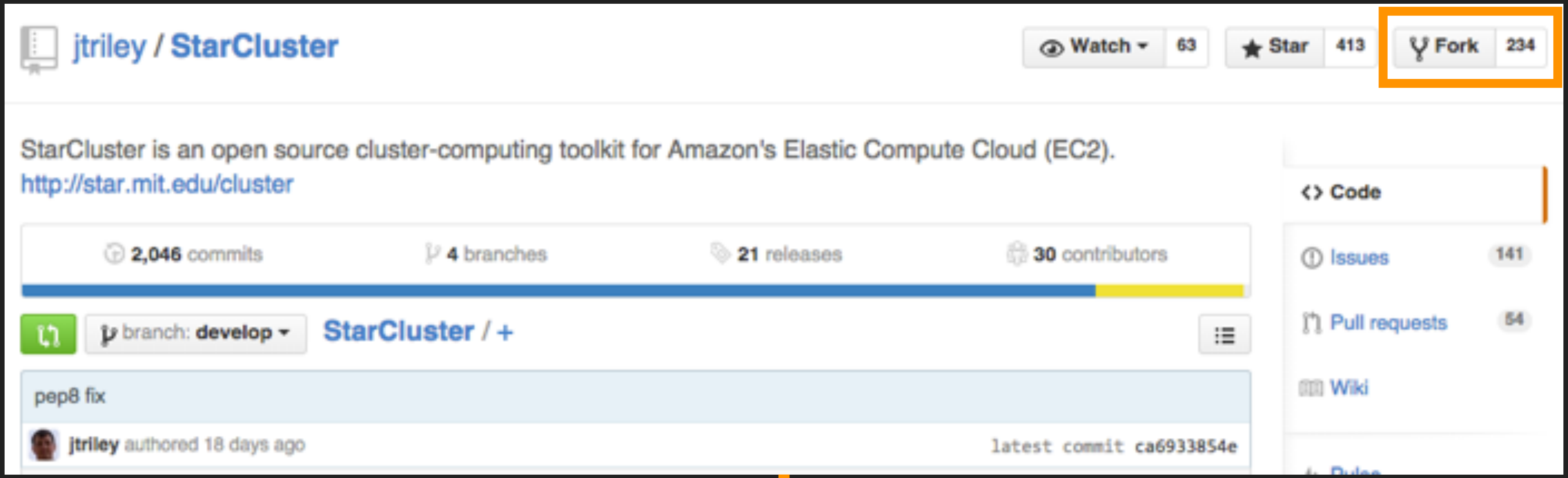
HTTPS clone URL

https://github.com/jtrile

You can clone with HTTPS, SSH, or Subversion.

Forking GitHub repositories

Forking: copies repo to your account, letting you work on a project you don’t have “push” privileges for.



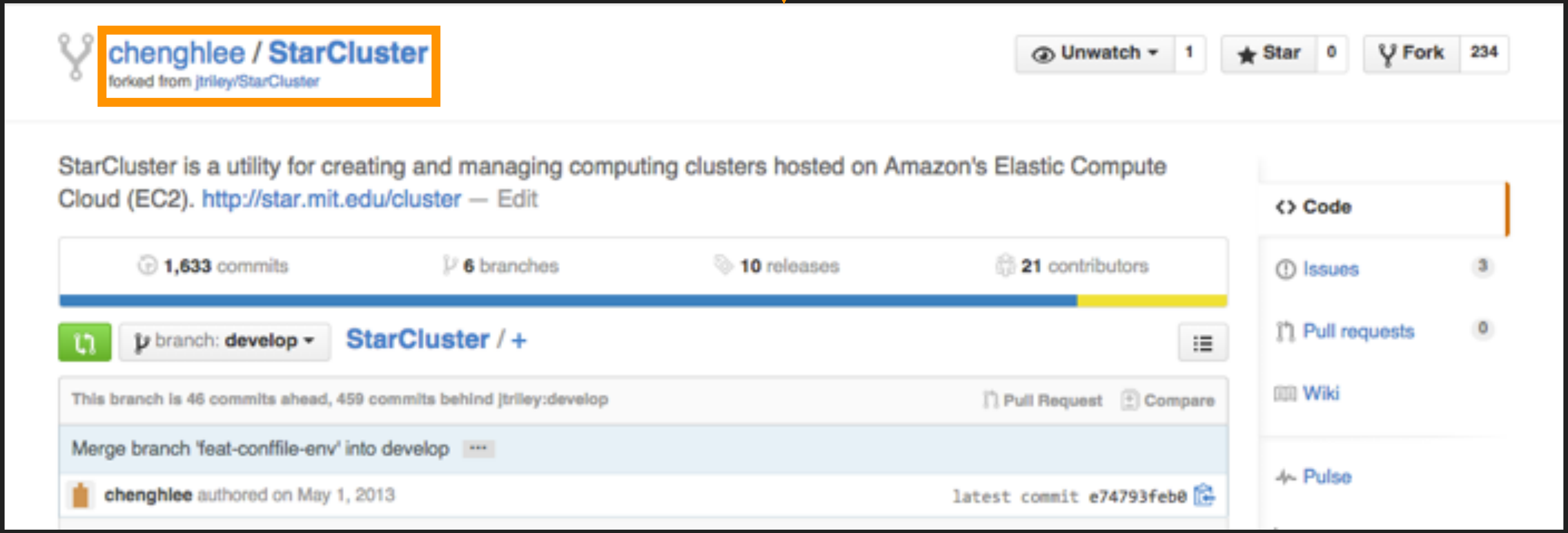
jtriley / StarCluster

StarCluster is an open source cluster-computing toolkit for Amazon's Elastic Compute Cloud (EC2).
http://star.mit.edu/cluster

2,046 commits | 4 branches | 21 releases | 30 contributors

branch: develop | StarCluster / +

pep8 fix
jtriley authored 18 days ago | latest commit ca6933854e



chenghlee / StarCluster
forked from jtriley/StarCluster

StarCluster is a utility for creating and managing computing clusters hosted on Amazon's Elastic Compute Cloud (EC2). http://star.mit.edu/cluster — Edit

1,633 commits | 6 branches | 10 releases | 21 contributors

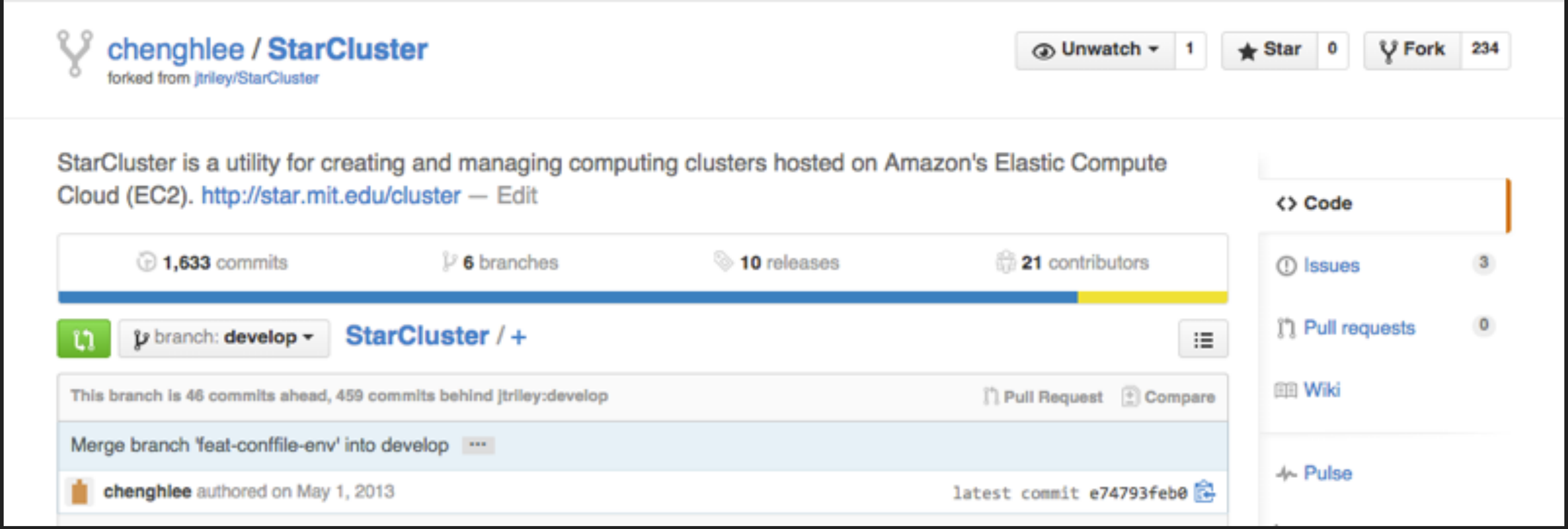
branch: develop | StarCluster / +

This branch is 46 commits ahead, 459 commits behind jtriley:develop | Pull Request | Compare

Merge branch 'feat-conffile-env' into develop
chenghlee authored on May 1, 2013 | latest commit e74793feb0

Forking repositories

Use your fork, *not the original project*, as the “origin” repository when working on your computer.



StarCluster is a utility for creating and managing computing clusters hosted on Amazon's Elastic Compute Cloud (EC2). http://star.mit.edu/cluster — Edit

This branch is 46 commits ahead, 459 commits behind jtriley:develop

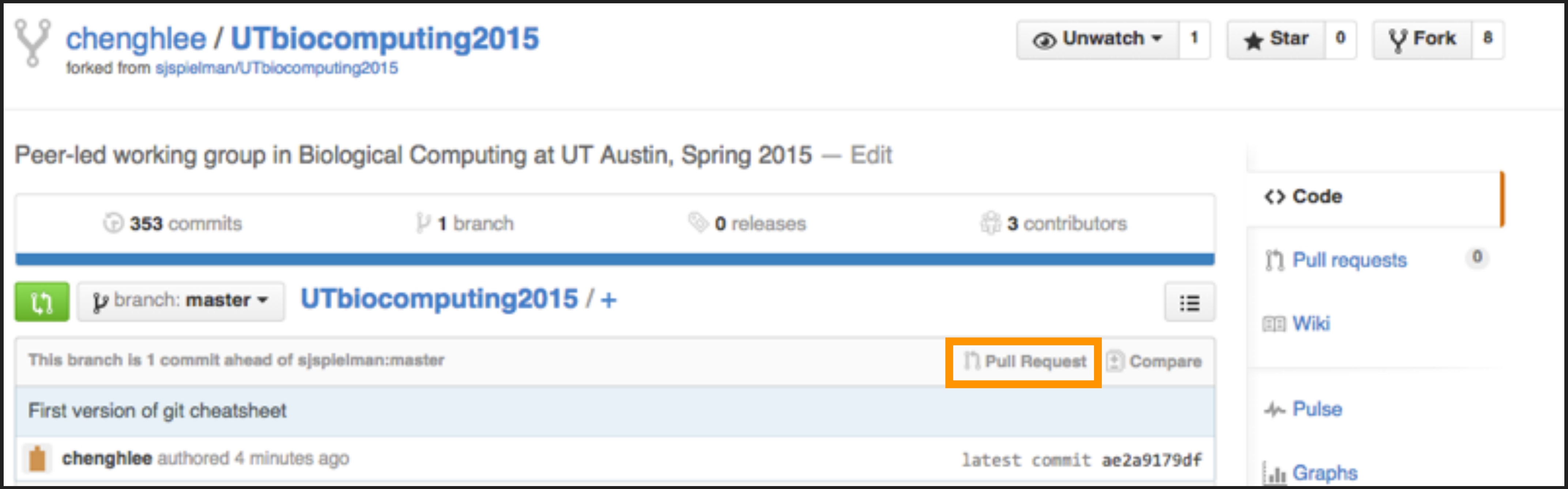
Merge branch 'feat-conffile-env' into develop

chenghlee authored on May 1, 2013

latest commit e74793feb0

Pull requests

Pull request: mechanism for contributing modifications from your fork back to the original project.



chenghlee / UTbiocomputing2015
forked from sjspielman/UTbiocomputing2015

Peer-led working group in Biological Computing at UT Austin, Spring 2015 — Edit

353 commits | 1 branch | 0 releases | 3 contributors

branch: master UTbiocomputing2015 / +

This branch is 1 commit ahead of sjspielman:master

First version of git cheatsheet

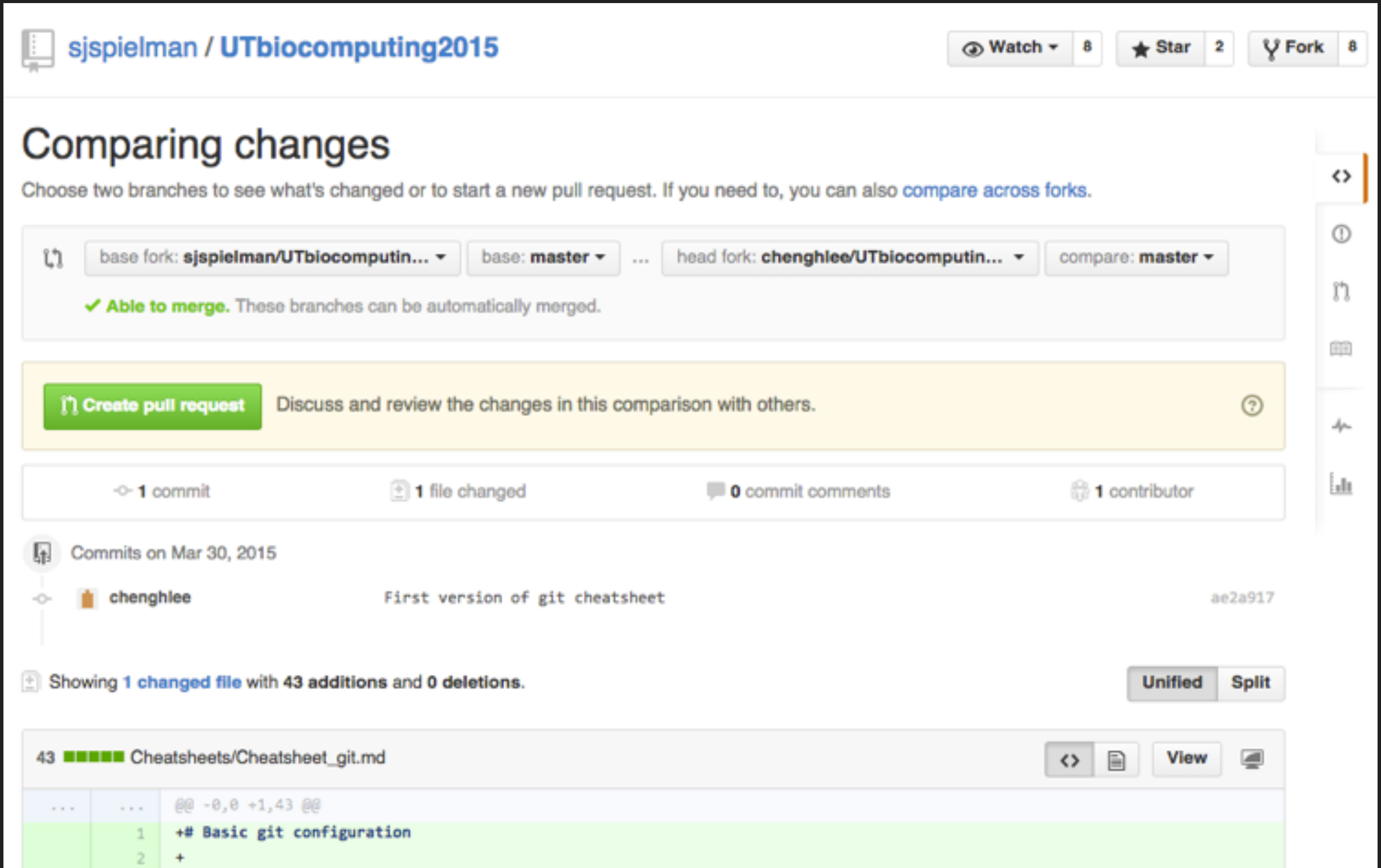
chenghlee authored 4 minutes ago   latest commit ae2a9179df

Initiated from your fork using the “Pull Request” button.

Note: Pull requests are a GitHub, not git, feature!

Pull requests

Pull request “dialog” lets you the repository and branch you want to send the change from and to.



sjspielman / UTbiocomputing2015

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

base fork: sjspielman/UTbiocomputin... base: master ... head fork: chenghlee/UTbiocomputin... compare: master

✓ Able to merge. These branches can be automatically merged.

Discuss and review the changes in this comparison with others.

1 commit 1 file changed 0 commit comments 1 contributor

Commits on Mar 30, 2015

chenghlee First version of git cheatsheet ae2a917

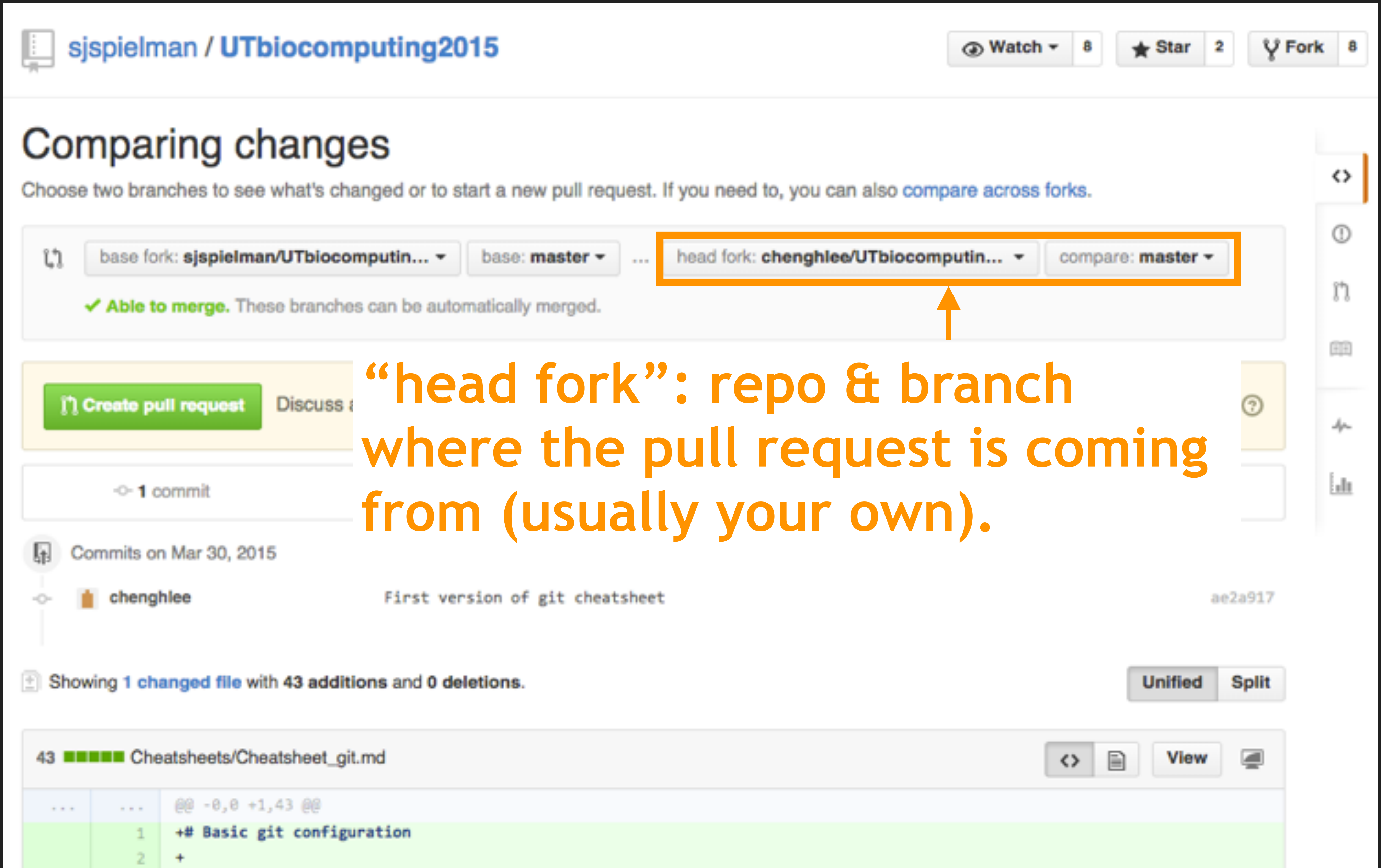
Showing 1 changed file with 43 additions and 0 deletions.

43 Cheatsheets/Cheatsheet_git.md

```
...    ...    @@ -0,0 +1,43 @@
         1    +# Basic git configuration
         2    +
```

Pull requests

Pull request “dialog” lets you the repository and branch you want to send the change from and to.



Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

base fork: sjspielman/UTbiocomputin... base: master ... head fork: chenghlee/UTbiocomputin... compare: master

✓ Able to merge. These branches can be automatically merged.

Commits on Mar 30, 2015

chenghlee First version of git cheatsheet ae2a917

Showing 1 changed file with 43 additions and 0 deletions.

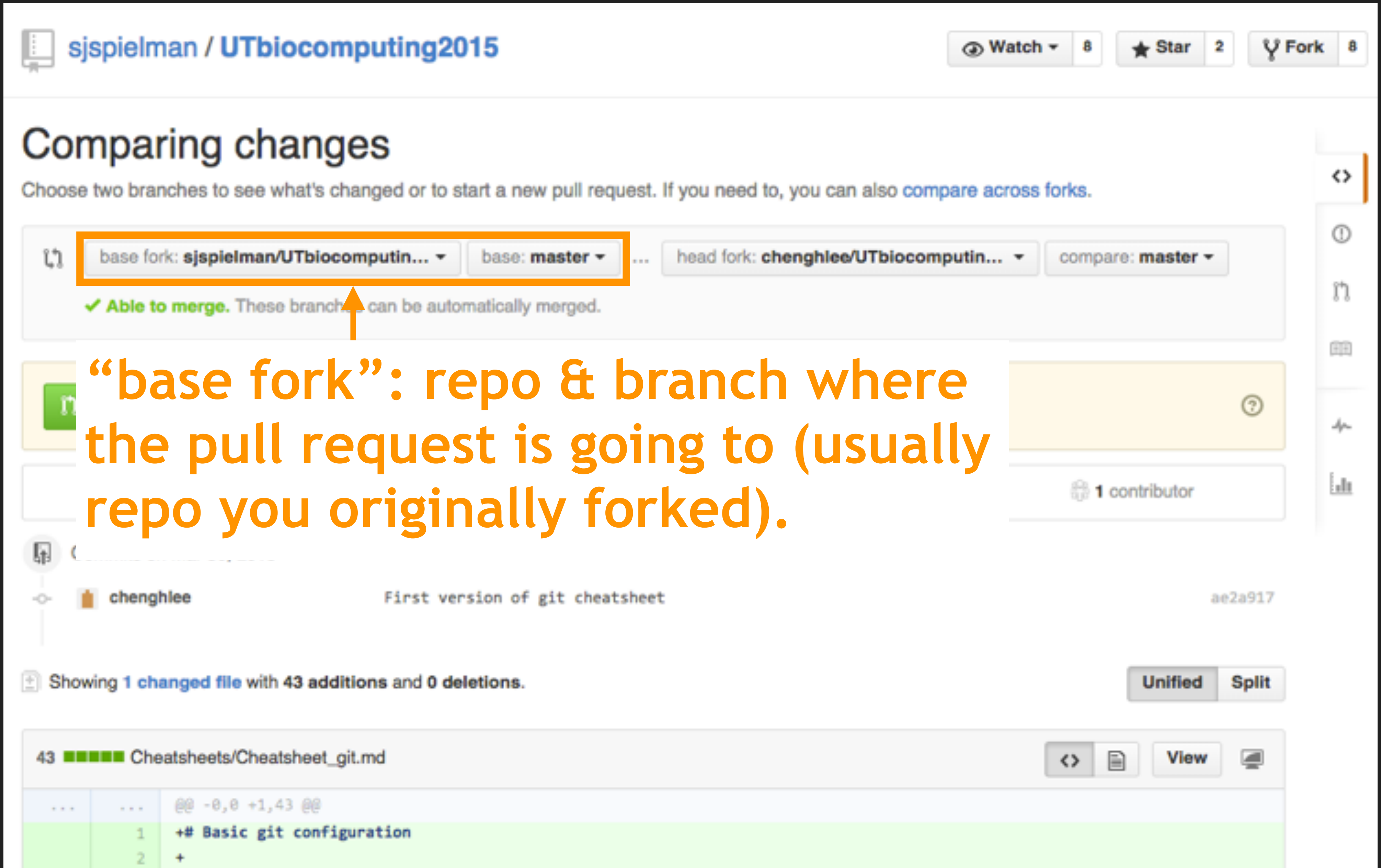
43 Cheatsheets/Cheatsheet_git.md

```
...  ...  @@ -0,0 +1,43 @@
      1   +# Basic git configuration
      2   +
```

“head fork”: repo & branch where the pull request is coming from (usually your own).

Pull requests

Pull request “dialog” lets you the repository and branch you want to send the change from and to.



sjspielman / UTbiocomputing2015

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

base fork: sjspielman/UTbiocomputin... base: master ... head fork: chenghlee/UTbiocomputin... compare: master

✓ Able to merge. These branches can be automatically merged.

“base fork”: repo & branch where the pull request is going to (usually repo you originally forked).

1 contributor

chenghlee First version of git cheatsheet ae2a917

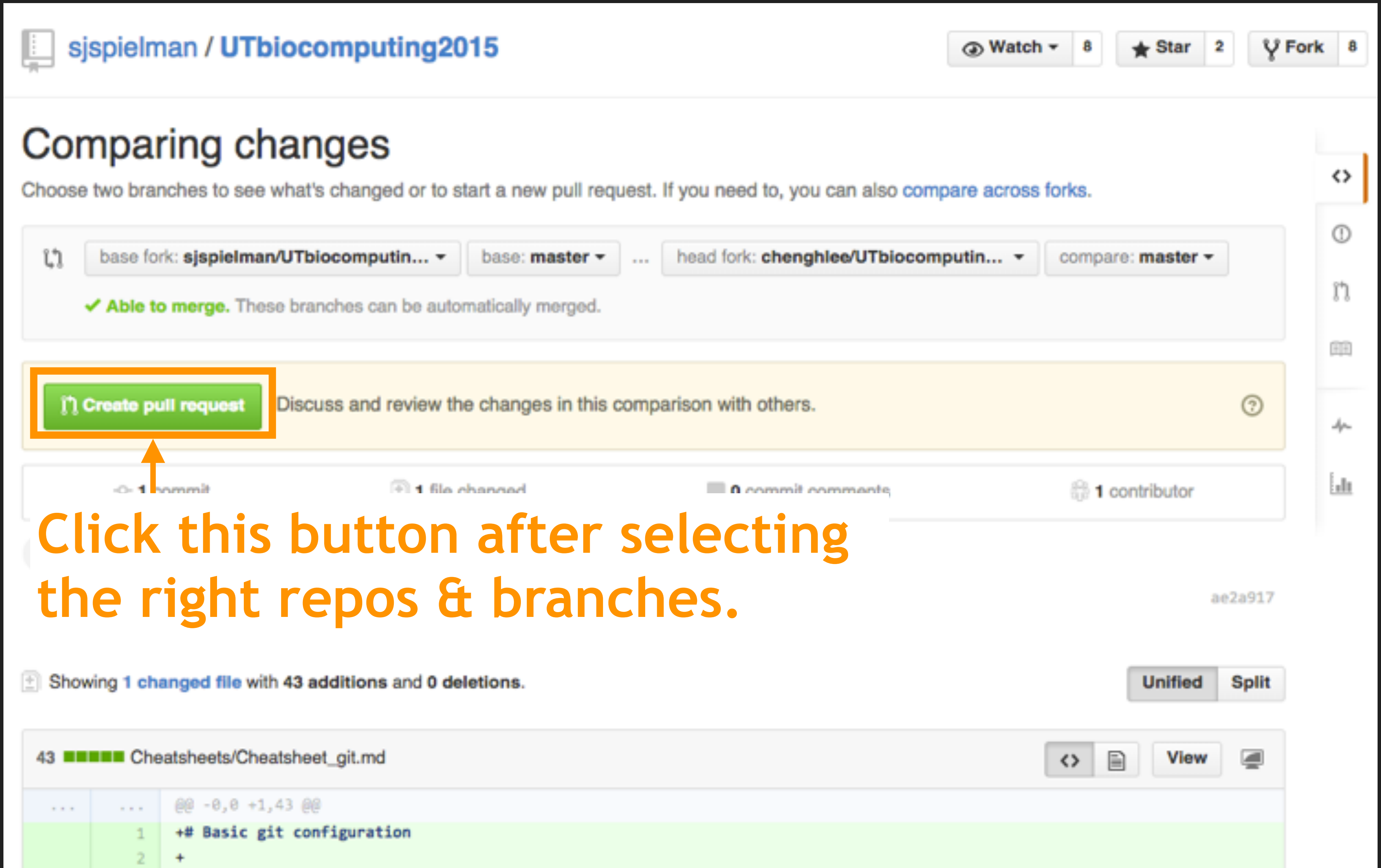
Showing 1 changed file with 43 additions and 0 deletions.

43 Cheatsheets/Cheatsheet_git.md

```
@@ -0,0 +1,43 @@
+# Basic git configuration
+
```

Pull requests

Pull request "dialog" lets you the repository and branch you want to send the change from and to.



sjspielman / UTbiocomputing2015

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

base fork: sjspielman/UTbiocomputin... base: master ... head fork: chenghlee/UTbiocomputin... compare: master

✓ Able to merge. These branches can be automatically merged.

Discuss and review the changes in this comparison with others.

1 commit 1 file changed 0 commit comments 1 contributor

ae2a917

Showing 1 changed file with 43 additions and 0 deletions.

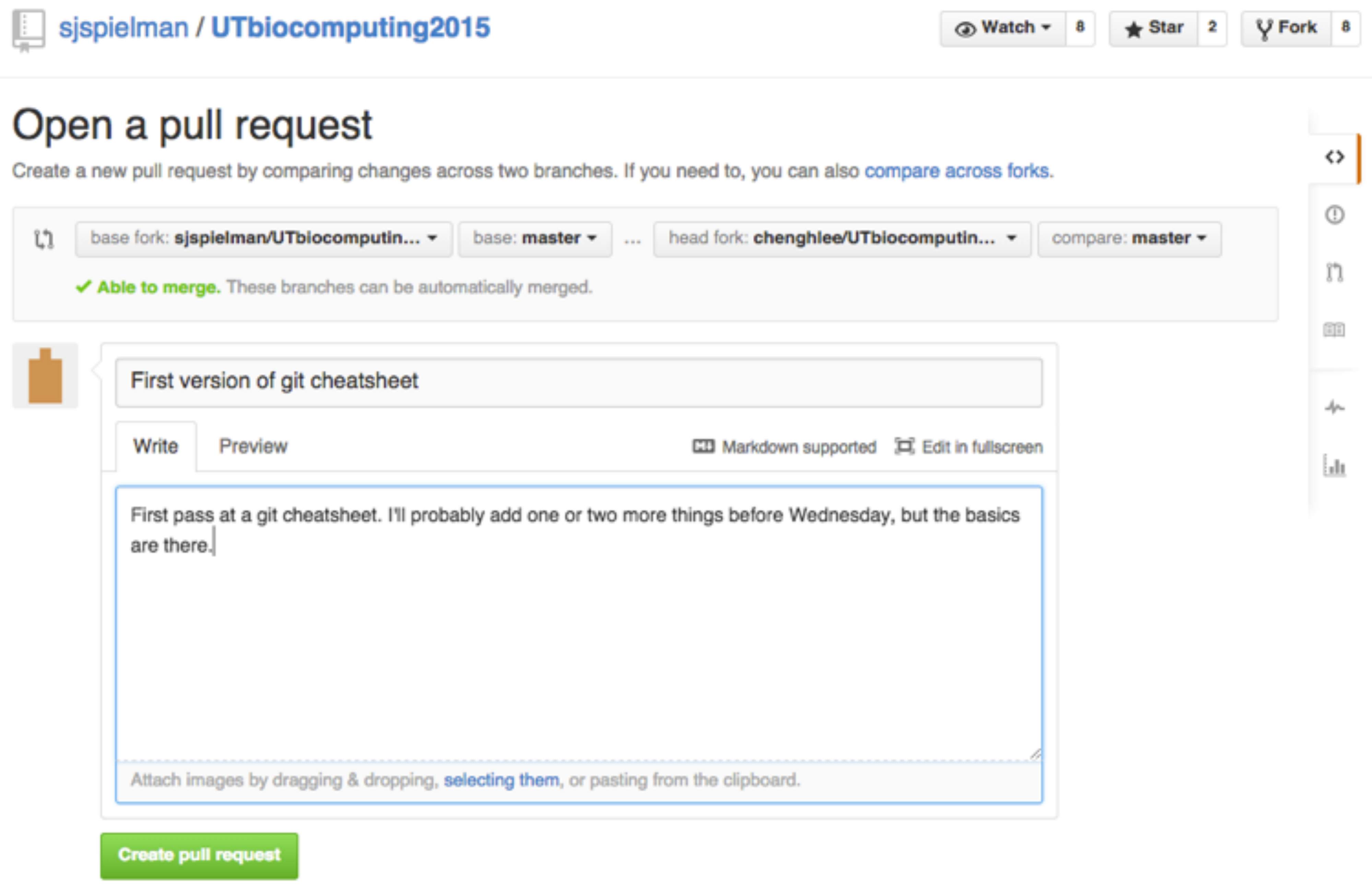
43 Cheatsheets/Cheatsheet_git.md

```
...     ...   @@ -0,0 +1,43 @@
          1   +# Basic git configuration
          2   +
```

Click this button after selecting the right repos & branches.

Pull requests

Last step in creating a pull request is to let upstream authors know what changes you're submitting.



Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.

base fork: sjspielman/UTbiocomputin... base: master ... head fork: chenghlee/UTbiocomputin... compare: master

✓ Able to merge. These branches can be automatically merged.

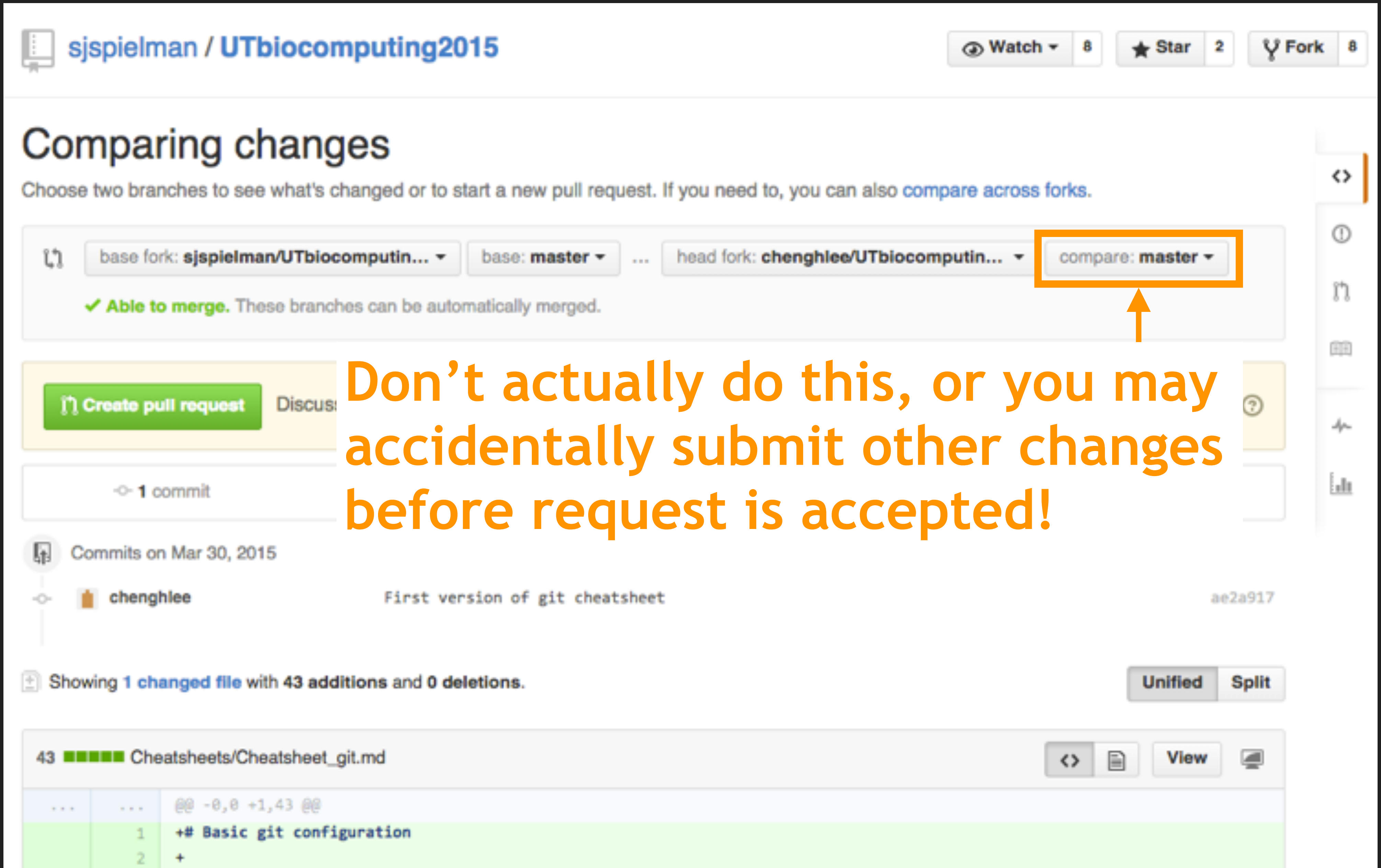
First version of git cheatsheet

First pass at a git cheatsheet. I'll probably add one or two more things before Wednesday, but the basics are there.

Attach images by dragging & dropping, selecting them, or pasting from the clipboard.

Pull requests

Tip: Generally better to isolate pull requests on separate branches, instead of sending them from master.



Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

base fork: sjspielman/UTbiocomputin... base: master ... head fork: chenghlee/UTbiocomputin... compare: master

✓ Able to merge. These branches can be automatically merged.

Don't actually do this, or you may accidentally submit other changes before request is accepted!

1 commit

Commits on Mar 30, 2015

chenghlee First version of git cheatsheet ae2a917

Showing 1 changed file with 43 additions and 0 deletions.

43 Cheatsheets/Cheatsheet_git.md

```
...  ...  @@ -0,0 +1,43 @@
       1  +# Basic git configuration
       2  +
```

Things not covered

This should be enough to get you started...

But git & most VCSes have other useful features; e.g.,

- Remotes & branches for complex dev. environments
- Tagging: labeling certain commits (e.g., “v1.0")
- Bug finding: bisect and blame
- Rebasing: rewriting history (use with extreme caution)

Also, not covered is working with large (open-source) projects:

- Managing hosting services like GitHub or BitBucket
- Integrating git & GitHub with other tools like bug trackers, automated testing frameworks, etc.

Odds and ends

Getting help:

- git help <command> (can be hard to understand)
- Git Book: http://git-scm.com/book
- StackOverflow

Visual tools (useful for managing commits and history browsing):

- Windows: TortoiseGit has tools built in
- OSX, Windows: SourceTree (http://sourcetreeapp.com/)
- Linux: gitk (pretty ugly though...)