A Laboratory Introduction to git

P. K. G. Williams (peter@newton.cx)

May 18, 2017
version `1e96911`

Introduction

Welcome to the git lab! This manual aims to help you learn the fundamentals of this awesome tool by walking you through some exercises that demonstrate its everyday functionality. Along the way we'll discuss some of its underlying principles and its connection to the rest of the Unix ecosystem too. We should note that we're assuming that you've already been given a broad overview of what git is, and why you might want to use it.

Learning git is like learning an instrument, or a language. There are certain concepts to master, but there's also just a lot of practice involved: repeating certain motions so that they become automatic. When it comes to computers, Unix pros talk about *finger memory*: I can type out `git commit -am` in my sleep. Of course, there's also the deeper appreciation of the underlying concepts and esoteric possibilities that you gain as you use a tool more and more. My assertion to you is that *time spent practicing git will more than repay itself in the future*. If you're just starting out as a programmer (*i.e.*, you've written less than 100,000 lines of code or so), some of its advantages are a bit hard to grasp; for what it's worth, the developer site GitHub has more than 4.5 million users and 6 million git repositories.

This lab is best done with a partner. Real-deal education research shows that it's much faster for you to learn something when talking it over with a partner rather than just staring at it by yourself. Getting back to the finger memory thing, though, please try to switch off between who's "driving" the keyboard and who's watching and commenting. It really does make a difference!

If you and your partner get stuck on something, trying asking the group next to you, or Google. Your lab assistants are happy to help you out, of course, but you'll learn more if you spend some time trying to solve problems on your own. That being said, with computers sometimes things go wrong even if you've done everything right — if you see a truly strange error message, particularly one associated with a non-git command, it's probably better to summon the lab assistants sooner rather than later. Deeper conceptual questions ("why" rather than "how") are also good to bring to the assistants.

Notation in this manual

Important terminology will be introduced in *italics*. Computer-y words will be written in a monospace font `like this`. Commands that you should type in a terminal in are presented with commentary this way:

```
$ echo hello world
```

Say hello.

You shouldn't type the leading dollar sign, which just indicates a generic terminal prompt. When the commentary is **in bold like this**, it means that a new command is being introduced.

Often we will give you only a piece of the command and you will have to complete it yourself. Sample text in **{bold braces}** must be replaced with a value that you choose or figure out for yourself, such as:

```
$ echo {the current year}
$ echo 2017
```

Say the year (sample as printed)
Say the year (what you actually type)

Don't type the braces! With the exception of the leading dollar signs and these pieces of sample text denoted with braces, *always type every command exactly as shown in the manual.* Take special care with punctuation and similar letters — command-line interfaces are finicky. The characters `'` (single vertical quote), `` ` `` (backtick), and `"` (double vertical quote) may look the same but they are all interpreted differently. Likewise for `l` (the letter ell), `1` (the digit one), `|` (pipe symbol), and so on.

The exercises are interspersed with questions so that you can check your understanding. They're represented thusly:

> *What is your name? What is your quest?*

Please actually write down the answers that you and your partner come up with!

There's one more bit of preliminary work to do before we get started in earnest.

Topic 1: git config

Before we start using git in earnest, we need to set up some important configuration. We can do this with the git program itself.

The git program that you run from the terminal is a sort of "Swiss Army knife" tool that provides many *commands* that do all sorts of different things. The command that sets up configuration is (quite sensibly) known as git config. Our first commands will set up some nice features that aren't turned on by default. Type the following commands in your terminal — once again, you should leave out the initial dollar sign:

```
$ git config --global color.ui auto          Use colors in printed output.
$ git config --global core.editor nano       Use nano to edit files.
$ git config --global alias.ci commit        A useful shorthand.
$ git config --global alias.s status         A useful shorthand.
```

Next we will tell git your name, which it will then embed in all of its logs of your activity. Remember that below, you should replace the sample text **{your name}** with your actual name. Remove the braces, but leave the double quotes.

```
$ git config --global user.name "{your name}"          Tell git your name.
```

You also need to tell git your email address. The value you enter here will be preserved permanently, so it is best to use a public address that will be long-lived.

```
$ git config --global user.email "{your email address}"          Tell git your email.
```

Now we can get started for real!

Topic 2: git clone

We'll get started by setting up a *git repository* to play with. This is the directory containing your actual content (*i.e.*, files) as well as git's supporting data.

There are two git commands to set up a repository: git init, which creates a new, empty repository; and git clone, which duplicates an existing one. We'll use the latter so that we have some files to work with right off the bat.

`$ cd`	Move to home directory.
`$ mkdir gitlab`	Create work directory.
`$ cd gitlab`	Move into it.
`$ git clone https://github.com/pkgw/bloomdemo.git`	**Clone an existing repository.**

Let's see what just happened. The sample text **{dirname}** below should be replaced with the name of a directory that the git clone command creates — the first `ls` that you run should reveal its name to you.

`$ ls`	Examine files.
`$ cd` **{dirname}**	Enter repository directory.
`$ ls`	Examine files again.
`$ ls -la`	Detailed listing of files.

By the way, in the final command above, the letter after the dash is an ell, not a one.

> *Where do git's supporting data seem to be stored?*

It's worth emphasizing that this cloned repository is both self-contained and self-sufficient. You can do anything you want to it without having to talk to GitHub again (without even needing an internet connection, in fact), and nothing you do will affect the GitHub version unless you explicitly attempt to synchronize the two. (Which is a vitally important process for collaborative work, but not something that we're going to worry about just yet.)

You can see a gory listing of git's housekeeping files with:

`$ find . -print`	**Print all file names.**

Diversion: Getting Help and the Pager

The `find` command above is a Unix program that you may not be familiar with. There will likely be more of these in the coming pages. If you'd like to learn more about a program, you can try reading its "manual page" with the `man` command:

`$ man find`	**Learn about `find`.**

When you run `man`, you enter a special subprogram called the *pager*. The pager is a tool for navigating lengthy textual output that offers more features than your terminal. Because `man`, git, and many other

Unix tools use it extensively, it is important to learn a bit about it. Different programs can do the job of the pager, but by default your system uses a Unix command called `less`. The `less` program is mainly controlled by commands that are single keystrokes. Some of these keys are:

Key	*Effect*
`q`	Quit the pager.
(Arrow keys)	Navigate as you'd expect.
(Spacebar)	Go forward a page.
`b`	Go backwards a page.
`<`	Go to the top.
`>`	Go to the bottom.
`/`	Search (type in query, then hit Enter).
`n`	Go to next search result.
`N`	Go to previous search result.
`&`	Filter (type in query, then hit Enter).
`h`	Print help information.

Since we don't want to get too distracted, just press `q` to quit the pager and return to your main terminal prompt.

The `man` program has its own manual page:

```
$ man man
```

Learn about `man`.

Unix manual pages are notoriously uneven in their quality. This is especially true regarding git itself, unfortunately. Google is often a better resource for beginners. The information on the StackExchange.com family of websites is usually very helpful.

You can run the `less` pager program like any other Unix command if you want to read a file right in your terminal:

```
$ less README
```

Read the bloomdemo repository's `README`.

Back on Track: Testing your Python

Let's get back to the git-ing! Well, almost. We should verify that your Python setup is working. If you run the following command, you should get a report about whether certain words are in the dictionary. You should *not* get any big honking error messages. If you do, please call over a lab assistant.

```
$ ./chkdict barn bern birn born burn
```

Check reality of words.

This program uses a technique called a *Bloom filter* to examine the words that you specify as arguments ("barn," "bern," etc.) and tell you whether they are found in the English dictionary. Bloom filters are unusual because they make mistakes in a one-sided way: if a word is in the dictionary, a Bloom filter will never say that it isn't; but sometimes a Bloom filter will think that a word is in the dictionary when it really is not. Why would you want that behavior? Bloom filters are much faster than filters that are always correct, and sometimes it makes sense to trade accuracy for speed.

Don't worry: the details of Bloom filters aren't important here, and we won't go into them any further. If you have extra time and are curious, read the file `bloom.py`.

Topic 3: git status, git checkout

Fundamentally, all git does is track changes to the files in your repository directory. It does this by comparing the files on disk, the *working tree*, with a recent snapshot of their contents. git stores many of these snapshots, each of which is called a *commit*. The most "recent" snapshot, in a sense, is known as the *HEAD* commit or just HEAD. Now, in some circumstances HEAD isn't chronologically the most recent, but for now that's the best way to think about it.

Let's modify one of the files that git's tracking in the working tree. We'll use the nano command to edit files. Your first task is to edit the file `chkdict` and change the string value that includes the words `"MIGHT BE"`. For now, just add the words `"... or it might not"` to the end of the string.

```
$ nano chkdict
$ ./chkdict barn bern birn born burn
```

Make a change.
Check that change stuck.

The next command we'll learn is git status, which tells you about your *local modifications* to the working tree compared to the most recent snapshot:

```
$ git status
```

Check modification status.

The git status command prints out several pieces of information — ignore most of it for now. But you should see git highlight that `chkdict` has been modified. It should also tell you how to discard your changes if you decide you don't like them. You do this with the git checkout command:

```
$ git checkout chkdict
$ ./chkdict {some words}
$ git status
```

Discard changes to chkdict.
Check that change is gone.
Check modification status.

Your change should be gone, and git should report that your working directory is *clean*. (If it's not, then it is, appropriately, *dirty*.)

> *Does git go so far as to restore the modification timestamp of `chkdict`? You can use `ls -l` to check.*

Let's pause for a second here. *The ability to discard your changes is in fact profoundly important and immensely powerful.* A *huge* amount of progress in programming and scientific research stems from one basic operation: "Let's try this and see if it works." If you have something that already works, the chief enemy of progress is the fear of breaking what you do have. *Git makes it safe to try things.* You can make any insane kind of change to a file in a git repository, and if you decide you don't like it, all you need to do is git checkout. You will truly have mastered the Zen of Git when you gleefully shred your most precious files without a care in the world.

These snapshots also mean that git is a nearly disaster-proof backup tool. If you have an up-to-date copy of your `.git` directory *somewhere*, you can probably recover your files.

```
$ rm dictbf.dat.gz                             Remove important file.
$ ./chkdict barn bern birn born burn           Check that program fails.
$ git status                                   Check modification status.
$ git checkout dictbf.dat.gz                   Restore deleted file.
$ ./chkdict barn bern birn born burn           Check that program is healed.
$ pwd                                          Check your directory.
$ rm -fv *.py *.pyc *.gz *dict* README         Type carefully!
$ ls -la                                       Not quite everything gone.
$ git checkout .                               Bring it all back.
$ ls -la                                       Huzzah!
```

The saved commits in the `.git` directory guard against many kinds of blunders. When you add in git's ability to synchronize repositories between different computers, you get almost complete invulnerability to loss of the data stored in the repository.

Of course, git needs to be tracking any file that you want to protect in this way. This doesn't happen automatically. The following commands create a new file called `mynewfile` and then delete it. Because we haven't told git to track `mynewfile`, we cannot recover it:

```
$ echo hello >mynewfile                        Create a new file.
$ git status                                   Check modification status.
$ rm mynewfile                                 Remove it.
$ git checkout mynewfile                       There's no saving that one.
```

Topic 4: git add, git commit

Since commits are the way that git remembers your files, we should learn how to make them! For reasons that we won't go into right now, making commits in git is a two-step process. First, you have to identify which changes you want to commit by *staging* them with the git add command. Then you actually create the new commit with git commit.

To demonstrate this, re-create the modification to `chkdict` that you did before:

```
$ nano chkdict                                 Change the "MIGHT BE" line.
$ ./chkdict barn bern birn born burn           Validate your change.
$ git status                                   Check modification status.
$ git add chkdict                              Stage the change.
$ git status                                   Note change in output.
$ git commit                                   Commit the staged changes.
```

When you run git commit you will be prompted to write a *commit message*. There are no rules about the message contents, but every commit must have one. Generally speaking, more sophisticated and organized projects will have more detailed policies about what should go in each message. For the kinds of projects you'll be working on, I suggest a one-line message of this form:

```
      chkdict: change the "MIGHT BE" message for fun
```

I find it helpful to identify the section or subsystem that the commit most strongly affects — that's the bit before the colon — and then tersely summarize what you did.

You *also* use git add to tell git to start keeping track of a new file. This is an example of a common annoyance with git: the same command (git add) will often do very different things, depending on how exactly you run it.

```
$ date >mynewfile                            Create a new file.
$ git status                                 Check modification status.
$ git add mynewfile                          Register it.
$ git status                                 Note change in output.
$ git commit -m "{your message}"             Commit the staged changes.
```

Here we've used a new option to git commit: the `-m` option, which lets you write the commit message right on the command line. You'll probably be mostly writing short messages, so the `-m` option can be a big convenience.

> *Say you had done the above steps through `git add newfile` and then decided you actually didn't want to commit the new file. What command would you run to reset things? (Hint:* git git status *is informative.) What happens to `newfile` in this case?*

There are a couple of other ways to "stage" changes to be committed. For instance, if we decide we don't want to keep `mynewfile` around anymore, we need to use git rm to register the removal:

```
$ rm mynewfile                               Remove our file.
$ git status                                 Check modification status.
$ git rm mynewfile                           Tell git we want to remove it.
$ git status                                 Note change in output.
$ git commit -m "{deletion message}"         Commit the staged changes.
```

An important point is that *git never forgets anything, so even if you delete a file from a repository, its contents are still stored and recoverable.* This is very important if, for example, you accidentally commit a password into a repository: even if you make a commit to remove the information, other people can still recover your password. There are ways to fully delete such information, but we won't get into them here.

There's also a git mv command that is the equivalent of an `add` and an `rm` together. Both this and git rm will perform the specified moves and/or removals on relevant working-tree files if they haven't already happened.

Finally, git commit has a useful option: the `-a` option, which automatically does the equivalent of git add on all of your modified files. It does *not* auto-add untracked files in your working tree. Using the standard Unix syntax for combining command-line options, we get a very useful pattern, exemplified here:

```
$ date >>README
$ git commit -am "README: add a timestamp"
```

Modify the README.
Add and commit.

Once you develop your finger memory, this is the quickest way to make commits.

Commentary: Commit Sizes and Messages

Most people starting out with coding and git'ing tend to evolve towards writing shorter and shorter messages for larger and larger commits, usually converging in one large commit made at the end of the day labeled "Update." I *strongly, strongly urge* you to try to get into the habit of committing in small chunks with thought-out log messages, even if it may take a while for the payoff to become clear.

The fundamental reason is that smaller commits are easier to understand. By breaking your work into smaller pieces, it's easier to reason about its correctness and overall design. This is true both as you write new code, and as you evaluate old code — every experienced programmer can tell you about revisiting *their own* year-old work and having no idea what they were thinking when they wrote it.

This path also goes both ways: the effort that you spend reasoning about how to break down your code into commits will help deepen your understanding of how to structure software in general. The more you do it, the easier it will get, and the better programmer you'll be.

Topic 5: git log

If you do find yourself wanting to review your previous commits, git log is the command to use.

```
$ git log
```

Show the commit history.

The git log command will open up the same pager program that we saw with the man command. But here, git log shows you a series of commits, each associated with an author, a date, and a message. Each commit also has a *commit identifier*, which is the string of 40 random-looking characters. You should see your own recent commits at the top of the output.

> *On what date was the very first commit made in this repository? You may want to consult the table of keystrokes that control* `less` *on page 5.*

> *Find a commit whose message contains the word "consuming." On what date was it made?*

An important aspect of git is that every distinct commit in the universe has a unique identifier. The identifiers look random but are uniquely determined by the commit's files and history.

> *(Optional.) Commit identifiers are 40-character hexadecimal strings, with each character having 16 possible values (0–9, a–f). If you made one new commit every second, about how many Hubble times*

> *would need to elapse before you used all possible identifiers? The Hubble time is about* 4.3×10^{17} *s, and* $2^{10} \approx 10^3$.

The git log command can also show you which files were changed in each commit relative to the one before. This is an *enormously* important aspect of git — is that it makes it easy not just to view commits, but to understand the *changes* that happened between different commits. The git log command presents change information in a compact form called a *diffstat* that we'll see more of later. It shows how many lines were added and removed from each file.

```
$ git log --stat
```

Show log, with change statistics.

> *In the most recent commit to modify the file* `INSTRUCTIONS`*, how many lines were added? How many removed?*

Topic 6: git show, git grep, git diff

While the git log --stat command is useful, the most common commands for examining changes between commits are git show and git diff. The first of these will show the set of changes associated with any commit in *diff format*, which is yet another convention that shows up throughout the Unix ecosystem. The format should be fairly intuitive to grasp, especially with the helpful colorful highlighting that git gives you:

```
$ git show 09933f
```

Show the named commit in the pager.

Above, we've named a commit based on the beginning of its hexadecimal identifier. In any given project, five or six hexadecimal digits is almost always enough to uniquely name a commit. (Optional exercise: how many different combinations of six hexadecimal digits are there?) You already know the name of another commit: HEAD.

```
$ git show HEAD
```

Show your most recent commit.

The diff for this commit will be fairly simple since it was a trivial example you authored just a little while ago.

> *Use* git log *to find the commit that adds code that uses* gzip *to compress the Bloom filter data file, then use* git show *to view the commit diff. What Python module is needed to add* gzip *support?*

Rather than showing an existing commit, the git diff command shows the difference between your working tree and *the staged set of changes — not* the most recent commit. We'll demonstrate this with a longer example. For the sake of pedagogy, please run through the following commands exactly as presented on your first time through this section.

The Bloom Filter False-Positive Rate

Read over the `chkdict` program. Towards the end you will see that it sets a variable named `fp`, which is the "false-positive rate" for the Bloom filter that it uses. The false-positive rate is the average frequency with which the Bloom filter will say that a word *is* in the dictionary when it really *isn't*. If `fp = 0.01`, the filter will think that 1% of non-words are actually words, on average. If `fp = 0.9`, the filter will think that 90% of non-words are actually words. (Once again, there's a tradeoff: filters with larger false-positive rates are less accurate but more efficient.)

Modify `chkdict` to print out this number before it reports the filter results for each word. To do this you should just need to add one line of code to the `chkdict` file.

```
$ nano chkdict                    Edit to print false-positive rate.
$ ./chkdict {some words}          Check everything works.
$ git diff                        Review unstaged changes.
$ git add chkdict                 Stage for committing.
$ git diff                        Review unstaged changes.
```

> *What is the reported false-positive rate?*

But wait a minute!

> *Given what you've been told, can the number that your program printed possibly be the correct false positive rate?*

Spoiler alert: No, it cannot.

The false positive rate came from a function called `fprate`. You can use git grep to locate its definition: this command searches for a string in the working tree files. (Technically git grep and the search feature of `less` use a Unix formalism called *regular expressions* or *regexes*, but for our purposes, you can just type what you're looking for.) If you locate the function definition and read the surrounding source code, you will see that the current implementation of the `fprate` function is just plain wrong.

`$ git grep fprate`	**Locate instances of "fprate".**
`$ nano` **{buggy file}**	Fix the bug.
`$ ./chkdict` **{some words}**	Check everything works.

What is the correct false-positive rate?

(Extra credit.) Use git checkout *to discard your fix, then the new command* git blame *to identify the commit that introduced the bug. Which one was it? When done, re-fix the bug. The additional new command* git help blame *may come in handy.*

The next set of commands will work through some of the permutations of having both staged and unstaged modifications in your working tree — recall that above we ran git add on our changes to `chkdict`, but didn't run git commit.

`$ git status`	Check modification status.
`$ git diff`	Review unstaged changes.
`$ git commit`	Commit staged changes.

What change(s) was/were just committed?

If you were to run git diff *now, after the* git commit*, what would you see? (Please try to guess the answer without just running the command!)*

`$ git diff`	Review unstaged changes.
`$ git add` **{remaining file(s)}**	Stage for committing.
`$ git diff --staged`	**Review staged changes.**
`$ git commit`	Commit staged changes.
`$ git status`	Check modification status.

Here, git diff --staged is a different mode that examines the differences between the *staged changes* and HEAD, while, as we've seen, plain git diff examines the differences between the *working tree* and the *staged changes*. If everything has gone well, you'll have a clean working tree, a new feature in `chkdict`, and a fixed bug. You can now muck about with the working tree however you want, confident that your important fixes won't be lost.

`$ pwd`	Double-check your directory.
`$ rm -fv *.py *.pyc *.gz *dict* README`	Type carefully!
`$ ls -l`	Confirm file removal.
`$ git checkout .`	Bring them all back.
`$ ./chkdict` **{some words}**	Verify correct FP rate is produced.

Topic 7: git branch, git checkout (redux)

If you run git without any arguments, you get a listing of its most commonly used commands:

```
$ git
```

Get help summary.

Taking a look at this list, we've touched on almost all of them! The next big step is to start getting into git's support for collaborative coding. The first thing we need to do is understand *branches*. And to understand those, we need to think about commits in a more careful, formal-math kind of way.

In this lab you've made several commits and reviewed them with git log. It would be reasonable to think of commits as coming in a time-ordered series: you start by making commit #1, then commit #2, then #3, and so on, for as long as you keep on updating your project. For very profound reasons, git does *not* identify commits this way. As we've seen, each git commit has a random-looking unique identifier. How do we know that one commit comes "after" another, then? In git this is accomplished by having each records the unique identifier of the commit that it derived from — its *parent*.

> *Try to think about this arrangement topologically. If you draw commits as circles with arrows pointing to their parents, what structures can you make that aren't simple linear chains? Keep in mind that we're introducing the concept of* branches*.*

It is true that in many cases your commits will form a linear chain in time order. However, they don't *have* to. In particular, you can create two different series of commits that diverge from the same parent — this is called *forking* or *branching*. For now, we won't really deal with *why* you might want to do this: please take my word for it that it'll be important to understand the underlying process. In fact, *the core technical breakthrough of* git *is its ability to deal with branches efficiently.* If you get the hang of git and then start dealing with branches in older tools such as Subversion, you'll see why git has been so revolutionary.

In git, a "branch" is just a name that refers to some specific commit, the *branch head*. That's all you need to reconstruct the entire project history back to its inception: the named commit embeds the unique identifier of its parent, which embeds the identifier of *its* parent, and so on. A branch's *history* is the set of all commits that have gone into it.

You can store data for many different branches at once, but there is only one *current branch*: the current branch is the one that the working tree and HEAD are synchronized with. The git branch command prints out the names of the branches in your repository:

`$ git branch`	**List branches.**

The current branch is denoted in the output of git branch with an asterisk.

> *Our repository has only one branch. What is its name?*

When you run git commit, git creates a new commit in its database that lists the current branch head as its parent, then it updates the record for the current branch to point to that new commit. These are both fairly straightforward operations. For instance, for each branch, the current identity of the branch head is stored in a simple text file:

`$ cat .git/refs/heads/master`	**Manually print the master branch commit id.**
`$ cat .git/HEAD`	Manually print out the branch that HEAD references.

(One of the reasons that git is so reliable is that commits involve *appending* new information to the repository but almost no *rewriting* of existing information, which is generally more dangerous. This is something to keep in mind when writing your own data-processing tools.)

It's simple to create new branches. Once you've done so, you can *switch branches* and alter the current branch with git checkout — another case of one command doing double duty, like git add before. For example, let's make a new branch called print-my-name. For now, this branch will be identical to the one you originally cloned from GitHub. If you have any uncommitted changes in your working tree, commit them or discard them before running the git checkout command — otherwise you may get errors.

`$ git branch print-my-name 90984a`	**Make new branch pointing to original files.**
`$ git status`	Check modification status.
`$ git checkout print-my-name`	**Switch to `print-my-name` branch.**
`$ git branch`	List branches.
`$ ./chkdict {some words}`	Verify that FP rate is not printed.

> *(Optional.) Use git log to verify that 90984a is the (shortened) unique identifier of the commit that you started with when you initially cloned the repository.*

Here, git checkout has done two things: it's updated information to say that the current branch is now the one named print-my-name, not master, *and* it's synchronized your working tree to match print-my-name. If you had any uncommitted changes, git checkout would have either preserved them or refused to run if it couldn't.

Let's create a commit on this new branch. We can do so using the same commands we've been using all along — because we've changed the active branch to print-my-name, that's the one that will be updated, not master.

We'll make another silly change hinted at in the name of the branch. Edit the top of the `chkdict` to add a line that prints your name after all of the `import` statements. Below, we've started assuming that you're getting the hang of things and don't need to see every git add and git commit command written out.

`$ nano chkdict`	Make the change.
`$` **{review and commit your change}**	Commit it.
`$ git log --oneline`	**Summarize history for current branch.**
`$ git log --oneline master`	Summarize history for master branch.
`$ git checkout master`	Switch to master.
`$ ./chkdict` **{some words}**	Verify that FP rate is printed.
`$ git checkout print-my-name`	Switch to print-my-name.
`$ ./chkdict` **{some words}**	Verify that your name is printed.

We've also slipped in another argument to git log, called --oneline, that produces a terser form of output.

You should see that the two branches start out with the same history (at the bottom of the log listings), but then *diverge*: they have commits in common, but both branches include commits that the other doesn't.

Topic 8: git merge

You might be able come up with reasons why you'd want to create multiple branches in a repository. For instance, you might maintain a piece of software that has two version series, 1.x and 2.x — even though your focus is on the 2.x version, you might still make updates to 1.x to support users who aren't able to upgrade. In these situations it's convenient to maintain two *long-lived branches*.

If that were the only use case, branching wouldn't be such a big deal. But git makes many things possible because you can not only create branches easily, but you can *merge* them back together later. Merging is a subtle concept but it is central to several common git operations, so it's important to take the time to understand how it works.

First, a technical perspective. Above, I wrote that each commit has a parent. That wasn't the whole truth. Really, each commit has *one or more parents*. When you merge two branches, you make a special commit that has *two* parents — a *merge commit*.

Mathematically, the history of a branch is therefore *not* necessarily just a linear sequence of commits going backwards from its head. The best way to visualize the history of a branch is actually as a mathematical *graph* — specifically, a *directed acyclic graph* or DAG. We won't get into DAGs here, but they're very useful reasoning devices for certain problems. DAGs are also the underlying formalism for the legendary Unix tool `make`. The tools of set and graph theory provide the formal foundation for the operations we'll describe below.

Now, let's think about what it *means* to merge two branches. It's best to focus on the ways in which the relevant branches differ, rather than what they have in common. Say, for instance, that I've started a project and ended up with two different branches derived from a common parent. One branch might be called fix-typos, and it might differ from the common parent in that it fixes some mistakes in the documentation. Another might be called add-feature, and it might differ from the common parent in that it adds an exciting new feature to the program. From this perspective, the merger of these two branches should clearly do *both* things: fix the typos *and* add the new feature.

Next, let's focus on the mechanics of what's actually happening with your files. We have emphasized that every commit corresponds to a snapshot of all of the files in your repository. To merge two different branches, we therefore need to combine two different snapshots. How does that happen?

In principle, you could examine the different snapshots corresponding to your two branches, identify the places where they differ from their common parent, figure out how to combine the changes, edit the files to combine the changes in both branches, and then git add and git commit the results. This would be an awfully tedious and error-prone process.

Fortunately, the git merge command automates this process just about as much as possible. For instance, let's merge your print-my-name branch into your master branch. Recall that master contains your change to print the correct Bloom filter false positive rate, while print-my-name doesn't, because we started it from the originally cloned snapshot. Instead, print-my-name has a change to print out your name in the `chkdict` program.

Peforming the merge is just a matter of running:

```
$ git checkout master
$ git merge print-my-name
```

Switch to master.
Merge this branch into master.

The git merge command will prompt you to write a message for the merge commit; unless you're working in a sophisticated project, you can usually just leave it with the default.

The above commands should have updated your master branch to contain not just your changes to print the Bloom filter false positive rate, but also your recent modifications to print your name. Running the merged code should demonstrate the effects of both sets of changes:

```
$ ./chkdict {some words}
```

Verify that both your name and FP rate are printed.

Hopefully that went straightforwardly enough. What happened under the hood is that git figured out what changes were contained in each branch relative to the common parent and decided that it could automatically apply both sets, *because the two sets of changes did not edit the same part of the same file*.

Resolving Merge Conflicts

If two different branches *do* modify the same part of the same file in different ways, you have a dreaded *merge conflict*. Because git doesn't actually know how your software works, the only way to resolve a conflict is through human intervention.

If a conflict arises, git leaves your working tree in a special funky state, with your files edited to point out where the conflicts are. You must decide how to resolve the conflicts, edit the files to implement the resolution, mark the files as dealt with using git add, and then finally git commit when everything is fixed. Things are a little bit simpler than you might fear since the various commands will tell you the necessary steps as you go through them.

We'll work through a merge conflict using some secret branches that came along with our clone:

```
$ git branch conflict-demo origin/master      New branch with pristine files.
$ git checkout conflict-demo                  Switch to conflict-demo branch.
$ git show origin/goodbye-option              View changes in the specified branch.
$ git show origin/no-skipmisses-option        Likewise.
```

Please pause here and think about what *changes* these two branches apply, relative to their common parent. The origin/goodbye-option branch adds a new option that prints "Goodbye" when the program is done running. The origin/no-skipmisses-option removes the `-s` option implemented in the original version of `chkdict`.

Now let's try merging both branches into conflict-demo:

```
$ git merge origin/no-skipmisses-option       Merge in no-skipmisses-option.
$ git merge origin/goodbye-option             Likewise.
```

At this point, git should report a conflict. The output of the commands isn't incredibly clear on this point, but both branches modified the same portion of the `chkdict` file, so git doesn't know how to proceed.

```
$ git status                                  Report merge/conflict state.
```

Behind the scenes, git has edited the file `chkdict` and added *conflict markers* indicating the problematic region that was edited in both branches. These consist of a long row of "<<<<", the final text found in one branch, then "====", the final text from the other branch, and finally ">>>>". Open up the file in nano and take a look:

```
$ nano chkdict                                Locate and examine conflict regions.
```

When a conflict happens, it's up to you to find *all* instances of these conflicts and edit the files to somehow do what *both* commits were attempting.

The example here tries to keep the conflict minimal. One branch adds a command-line option, and another branch removes one, so they both edit the same stanza of option-handling code at the beginning of `chkdict`. To resolve the conflict, you first have to decide what the merged code *should* do — it seems clear that it should accept the new argument and remove the old argument. You then need to *implement* that solution by editing the code. To implement the solution you should use the sample text from the two branches as a reference, but the best implementation might not look exactly like *either* version. After you're done, all of the conflict markers should be gone from your files. You can then git add and git commit as usual — git uses a special internal reminder to realize that this commit represents a merge and not just a regular commit.

```
$ nano chkdict                                Fix the conflict.
$ git add chkdict                             Stage the fix.
$ git diff --staged                           Examine the changes of the final merge.
$ git commit                                  Commit the fix.
$ git log                                     Review commit history.
```

> *What does the commit history of the* `conflict-demo` *branch now look like, as a graph?*

Commentary: Well-definedness of merges

Some merges are not semantically possible. If one branch of a command-line program adds an option called `-s` to sort output, and another adds an option called `-s` to save inputs, there's just no way to merge the two branches while preserving their intent. When you make a merge commit, you're *asserting* to git that it preserves the meanings of the two merged branches in some reasonable way, but it's impossible for git, or any other objective analysis, to *prove* that this is the case. In fact, you could "resolve" a merge conflict by replacing all the contents of every file in your project with Doctor Who fan fiction.

What underlies the way that git's merge system works is an assumption that *modifications to different parts of different files are probably independent* — it will only flag a conflict if two commits modify the *same* part of the *same* file. But this assumption can fail. For instance, I could merge two branches where one renames a variable in file A, while the other references the old variable name in file B. There's no way to detect the problem without running the code. More broadly speaking, *a merge is not a well-defined, mechanical operation, and it can never be entirely automated.* Fortunately, in the real world, the different-places-means-independence assumption is generally true, and most merges can proceed mechanically.

Stepping back, the same general point is true of any kind of commit. Commits in a sequence are generally assumed to be related and not include enormous changes. But there's no reason that I can't make a single commit that deletes all of the source code to my project and replaces it with Doctor Who fan fiction. Fundamentally, every git commit both encodes something very precise and something very nebulous. The precise thing is an exact, reproducible snapshot of a set of files in a repository. The nebulous thing is an assertion by a human that those files bear some useful relation to the ones found in the commits that came before.

Topic 9: git remote, git fetch, git pull

Finally we get to the payoff. *The features of git's branching system are sufficient to allow collaborative work on a project by a distributed, decentralized team.* Or, more prosaically, git's ability to synchronize repositories between computers builds on its branching and merging infrastructure. So even if you're not planning on contributing to the Linux kernel any time soon, a solid understanding of the branching system is important for synchronizing your work between different machines and sharing your work with others.

Each git repository stores a list of other repositories that it knows about, known as *remotes*. These can be in a different directory on the same machine, or accessible over the network using any of several protocols. You can download updates from a remote and, as we'll see in the next section, also *push* updates to it. The simplest way to collaborate using git is to set up a central repository on the network that everyone can push to — which is exactly what GitHub does and why it's so popular. Other, less-centralized models are possible, but we won't discuss them here. It bear emphasizing, however, that a key advantage of git is that if GitHub were to disappear tomorrow, we wouldn't lose any data, because each repository is self-sufficient, and by setting up new remotes we could start collaborating again almost seamlessly.

Every cloned repository starts with a remote called origin, which you've probably been seeing mentioned by various tools over the course of the lab. You can learn about your remotes with, unsurprisingly, the git remote command.

```
$ git remote
$ git remote show origin
```

List named remotes.
Show details about `origin`.

Each remote is associated with *remote branches*, which the git remote show command just listed for us. The git branch command will also list them if you give it the -a ("all branches") option:

```
$ git branch -a
```

Show all branches.

You can't make commits on these branches yourself. However, you can download updates from the remote with the git fetch command. We can run it here, but since I haven't craftily updated origin since this lab started, all you'll see is silence, indicating that there are no new commits:

```
$ git fetch origin
```

Update remote branches for origin.

If there *were* new commits, what would you do? Well, you've got a branch that you're working on (generally master, or conflict-demo here), and you've got one with commits that you want to import (origin/master — note that the remotes/ prefix isn't necessary). That sounds like time for a git merge!

```
$ git checkout conflict-demo
$ git merge origin/master
```

Make sure we're on the right branch.
Merge in (nonexistent) updates.

The command git pull essentially does a git fetch followed by a git merge, though it has a few bells and whistles.

```
$ git pull origin master
```

Fetch and merge more nonexistent updates.

We haven't discussed this, but it can be very difficult to work through merges if your working tree is dirty. Therefore, to synchronize your repository with the outside world, you should generally follow this procedure:

1. Run git status to see if you have any uncommitted changes that need to be tidied up.

2. If you do: either commit your work or decide to throw it away; or otherwise clean up your working tree.

3. Finally, run git pull to fetch and merge any updates from your collaborators.

Topic 10: git push

Finally, there's the matter of *pushing* updates to a remote repository. In practice you'll generally be doing this by pushing updates to GitHub, but to keep the lab self-contained we're going to do everything on your local disk. git does a good job of abstracting between different kinds of remotes, so the underlying steps are exactly the same in either case.

```
$ mkdir ../sync.git                           Create sync repo directory.
$ cd ../sync.git                              Move into it.
$ git init --bare                             Create push-able repo.
$ ls                                          Examine files.
$ cd ../bloomdemo                             Back to working tree.
$ git remote add localsync ../sync.git        Register the new repo.
$ git push localsync master                   Publish our master branch.
$ git push localsync print-my-name            Publish our print-my-name branch.
$ git remote show localsync                   Show details about localsync.
```

This sequence of commands creates a local repository that we can push to. The `sync.git` directory is a *bare repository*: it has commit data, branches, and a HEAD, but no actual working tree! You can't push to a repository with a working tree since git would need to be able to rewrite all the working files, which it can't safely do if there happen to be local modifications. The files in the `sync.git` directory are the same as those in your working tree's `.git` directory. It's conventional to give bare repositories names ending with `.git` even though they're directories.

The git push command has somewhat-sophisticated syntax allowing you to remap branch names between repositories. For instance, to publish our `master` branch under the name `experiments` on `localsync`, you write:

```
$ git branch -a                               List all branches.
$ git push localsync master:experiments       Publish master as experiments.
$ git branch -a                               List all branches.
$ git remote show localsync                   Show details about localsync.
$ git push localsync :experiments             Delete experiments on localsync.
$ git remote show localsync                   Show details about localsync.
```

> *What's different the second time you run* git branch -a*?*

As we see above, the special syntax of pushing a blank branch name ("`git push localsync :experiments`") deletes the branch on the remote. Having successfully pushed our changes, we can now shred our files without worrying.

`$ cd ..`	Move to `gitlab` directory.
`$ pwd`	Confirm your directory.
`$ rm -rfv bloomdemo`	Destroy all of your work!
`$ ls -la`	Confirm it's gone.
`$ git clone https://github.com/pkgw/bloomdemo.git`	Clone an existing repository.
`$ cd bloomdemo`	Move into it.
`$ git remote add localsync ../sync.git`	Register our backup.
`$ git branch -a`	List all branches.
`$ git fetch localsync`	Fetch data from localsync.
`$ git branch -a`	List all branches.
`$ git merge localsync/master`	Merge changes from backup.
`$ ./chkdict` **{some words}**	Confirm our changes.

With a few short commands, you were able to recover all of the work that you did during this lab — not just the files themselves, but the *history* of what you did to them — even though we completely erased the directory in which you did all of the work. I hope you're impressed!

Recap

That's the end of the lab! What should you take away from this all?

- From one angle, git is an excellent *backup tool* for your files.
- From another, it's a *freedom tool* that lets you experiment in your projects, secure in the knowledge that you can reset things to a known-good state if you decide that you messed up.
- It's also an amazing *collaboration tool* that provides a tractable way for groups of people to work together on projects in a decentralized manner.
- It is also, admittedly, a *complicated tool* with many esoteric features and a sophisticated underlying theoretical model. We've barely scratched the surface of its capabilities.

If you don't want to be chained to a paper copy of this lab manual, is there an electronic form? There is, and it's tracked in git, of course:

https://github.com/pkgw/git-lab

git Command Quick Reference

There are many commands that are not listed, and all of these commands can do much more than is given in the summaries below.

Command	*Purpose*
`git add` **{files}**	Stage files for committing, or register new files.
`git branch` **{name}** **{initial}**	Create a new branch pointing at initial.
`git checkout` **{branch}**	Switch to a new branch.
`git checkout` **{file}**	Restore a file to its HEAD state.
`git clone` **{URL or path}**	Clone an existing repository.
`git commit`	Make a new commit.
`git diff`	Show changes between working tree and staged changes.
`git diff --staged`	Show changes between staged changes and HEAD.
`git fetch`	Fetch updates from a remote.
`git grep` **{regex}**	Search for text in the repository contents.
`git init`	Create a new empty repository.
`git log`	Show commit logs.
`git merge` **{branch}**	Merge another branch into the current one.
`git mv` **{old}** **{new}**	Rename a git-tracked file.
`git pull`	Combination of `fetch` and `merge`.
`git push`	Publish updates to a remote.
`git rm` **{file}**	Delete a git-tracked file.
`git show` **{commit}**	Show the changes in a commit.
`git status`	Report status of the working tree.