A Laboratory Introduction to git and GitHub

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

October 17, 2022

version `c5ca452`

Typeset in EB Garamond and Adobe Source Code Pro using LaTeX.

Introduction

Welcome to the git lab! This manual aims to help you learn the fundamentals of this awesome tool by walking you through its everyday functionality. You'll also learn about the popular commercial website GitHub, and how the two fit together.

This manual includes a lot of marginal notes. They are intended to provide extra context and information for reference, but you don't need to read them for the core lab activities.

This is an example marginal note!

Here are some basic guidelines for this lab:

- *We strongly recommend that you actually type out the commands shown here*, and not just read the commands passively.
- *This lab is best done with a partner.* However, please try to switch off between who's "driving" the keyboard and who's watching and commenting.
- *Ask for help!* That's the whole point of doing this lab as a group instead of on your own. If you and your partner get stuck, trying asking the group next to you, or Google.
- Of course, 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, get the attention of an expert.

Learning git is like learning an instrument or a language. There are certain concepts to master, but there's also just an element of practice. Unix pros talk about *finger memory*: you need to type a command a few hundred times to start really using it fluently.

Notation in this manual

Important terms will be introduced in *italics*. Computer-y words will be written in a monospace font `like this`. Commands to type in the terminal in are presented with commentary like so:

```
$ echo hello world                                          Say hello.
```

Don't type the dollar sign, which indicates the terminal prompt.

Often you'll have to fill in part of a command yourself. Substitute **{sample text}** with a value that you

choose or figure out for yourself. For instance:

```
$ echo "{the current year}"          What's printed in this manual.
$ echo "2022"                        What you actually type.
```

Don't type the braces!

Take special care with punctuation and similar letters — command-line interfaces are finicky. The characters `'` (single vertical quote), `` ` `` (backtick), and `"` (double vertical quote) may be similar but they are all interpreted differently by the computer. Likewise for `l` (the letter ell), `1` (the digit one), `|` (pipe symbol), and so on.

With the exception of the leading dollar signs and **{sample text}**, you'll get your best results if you *type every command exactly as shown in this manual*.

Other actions you that you should perform on your computer are typeset like so:

Start your code editor.

This manual also includes questions to can check your understanding. Let's start simple:

> *What is your name?*

No one's going to grade you, but it will help learning if you actually think about these questions and write down your answers!

Part 1: Command-Line Basics

Topic 1: git config

Before we really get started, we need to set up some important configuration. We can do this with the git program itself. Run these commands:

```
$ git config --global color.ui auto          Use colors in printed output.
$ git config --global alias.ci commit        A useful shorthand.
$ git config --global alias.s status         A useful shorthand.
```

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) git config.

For the next two, remember to replace the sample text with real answers. Don't type the braces, but do type the double quotes.

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

git will embed your name and email address in all of its logs of your activity, so it's important to declare them! The values you enter here will be preserved permanently. For email, it's best to use a long-lived, public address.

Now we can get started for real!

Topic 2: git clone

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

```
$ 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 repo.
```

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 use the latter so that we have some files to work with right off the bat.

What file(s) did git just create?

```
$ ls                                                  Examine files.
```

Let's navigate into this directory and examine its contents.

```
$ cd bloomdemo                                  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.

> *The detailed file listing should reveal a hidden directory, used by* git *to store its supporting data. What is this directory called?*

Your 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.

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

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

Finally, to prepare for some concepts that come later in this lab, run this command:

```
$ git branch print-my-name                      We'll come back to this later.
```

Testing your Python

The repository we cloned is named bloomdemo. Let's learn a bit more about it:

```
$ cat README                                    Print the bloomdemo repository's README.
```

The README says that this repository contains Python code. Before you go any farther, let's check that your Python installation works:

```
$ ./chkdict barn bern birn born burn            Check reality of words.
```

When you run the above 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 consult with an expert.

The code here uses a tool called a *Bloom filter* to let you check whether a words is found in the English dictionary. Bloom filters make mistakes, but 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 isn't. Why would you want that behavior? Bloom filters are much faster than filters that are always correct, and sometimes it's good to trade accuracy for speed. Don't worry: the details aren't important here. 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. It compares 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 see how this process works.

> *Open the file* chkdict *in your code editor. Find the line that includes the words* `"MIGHT BE"`*. Edit it to add the words* `"... or it might not"` *within the quotation marks. Save the file when done.*

Not sure how to open files in a code editor? Ask your partner or one of the experts!

If everything worked, the output of the program should change when you rerun the program:

```
$ ./chkdict barn bern birn born burn                    Check that change stuck.
```

Next we'll use git status to learn about *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 them 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 can do this with the git checkout command:

```
$ git checkout chkdict                                  Discard changes to chkdict.
$ ./chkdict {some words}                                Check that change is gone.
$ git status                                            Check modification status.
```

Newer versions of git use git restore instead of git checkout here. The new name was created because git checkout did two very different jobs, as we'll see later. But not that many computers have a new enough version of git installed just yet.

Your change should be gone, and git should report that your working directory is *clean*.

If your working directory isn't clean, 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.*

If you have something that already works, the chief enemy of progress is the fear of breaking what you do have. You will truly have mastered the Zen of Git when you gleefully shred your most precious files without a care in the world.

Let's pause here. *The ability to discard your changes is profoundly important and immensely powerful.* A *huge* amount of progress stems from one basic operation: "Let's try this and see if it works." *Git makes it safe to try things* because it makes it easy to revert to known-working code.

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!
```

Many developers have a saying that inverts the point of view: *if it isn't in Git, it didn't happen*.

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.

Topic 4: git add, git commit

If you want to protect a file, git needs to be tracking it. 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.
```

Commits are the way that git remembers your files, so we should learn how to make them! 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. We'll demonstrate this.

We won't go into *why* this two-phase approach was chosen, but there are good reasons.

Open the file chkdict *in your code editor and recreate the change you made before. Save the file when done.*

Now run these commands:

We'll see below that git add can also teach git about new files. This is an example of a common annoyance with git: the same command will often do very different things depending on how exactly you run it.

```
$ ./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.
```

How does git *describe the status of* `chkdict` *after you have "added" it?*

To finalize the commit, run:

```
$ 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. Here, I suggest a one-line message of this form:

Generally speaking, more sophisticated and organized projects will have more detailed policies about what should go in each message. 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. Additional lines should report *why*. The rule of a gold-plated commit message is that someone knowledgeable should be able to recreate your changes based only on the message.

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

You *also* use git add to tell git to start keeping track of a new file.

```
$ 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 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.
```

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.

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*.

There's also a git mv command. It's 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                                  Modify the README.
$ git commit -am "README: add a timestamp"       Add and commit.
```

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

Topic 5: git log

If you 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 a "pager" program as described in an Appendix. It 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 the pager on page 27.*

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

Every distinct git commit in the universe has a unique identifier. The identifiers look random but are uniquely determined by the commit's files and history.

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.

(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* $2^{10} \approx 10^3$.

This is an *enormously* important aspect of git — it makes it easy not just to view commits, but to understand the *changes* that happened between different commits. GitHub's web interface makes these things even easier.

The git log command can also show you which files were changed in each commit relative to the one before. This *diffstat* mode 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 diff, git grep

The diff format is yet another convention that shows up throughout the Unix ecosystem.

The command git show will show the set of changes associated with a commit in *diff format*. It 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.
```

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?

Above, we've named a commit based on the beginning of its hexadecimal identifier. You already know the name of another commit: HEAD.

```
$ git show HEAD                     Show your most recent commit.
```

The diff for this commit is 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?*

The Bloom Filter False-Positive Rate

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.

Open the `chkdict` *file in your code editor.*

Towards the end of the file you will see that it sets a variable named `fp`. This variable is the "false-positive rate" for the program's Bloom filter.

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. This parameter is useful because there's a tradeoff: filters with larger false-positive rates are less accurate but more efficient.

Modify `chkdict` *to print out* `fp` *before it reports the filter results for each word. To do this you need to add one line of code to the* `chkdict` *file. Save the file when done.*

```
$ ./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! A false-positive rate is a probability. Your program should have printed out a negative number, which is not a legal probability.

Technically git grep and the search feature of `less` use a Unix formalism called *regular expressions* or *regexes*. These are powerful and cool constructs, but for our purposes, you can just type what you're looking for.

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.

```
$ git grep 'def fprate'                                  Locate instances of "fprate".
```

The repository that you checked out has had a bug intentionally inserted into this function.

Open the file identified by git grep. Find the definition of the `fprate` function. Remove the bug according to the instructions in the file.

```
$ ./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 change to make `chkdict` print out the false-positive rate, but we 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? 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. 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.

This contrasts with plain git diff, which examines the differences between the *working tree* and the *staged changes*.

```
$ 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.
```

Part 2: Collaboration: Foundations

Topic 7: git branch, git checkout (redux)

A commit name is all you need to reconstruct the entire project history back to its inception: the named commit embeds the unique identifier of its parent(s), which embeds the identifier of *its* parent(s), and so on. A branch's *history* is the set of all commits that have gone into it.

You were just told that in git, a "branch" is just name that refers to some specific commit, the *branch head*. 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?*

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. When it does rewrite a file, like the branch head file, the operation is isolated and minimal. This is a design practice to keep in mind when writing your own data-processing tools.

When you run git commit, git creates a new commit in its database, then updates the the current branch to point to that new commit. The current identity of the branch head is stored in a simple text file:

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

Creating new branches is simple. In fact, we've already done it — remember how we ran git branch print-my-name back when you checked out the repo? Now we'll follow up on that and *switch branches* to activate it with git checkout — another case of one command doing double duty, like git add before. 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.

In newer versions of git, this use of git checkout is superseded by git switch.

```
$ 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.
```

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 main, *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.

> *Use* git log *to look at the history of your current branch. The commits that you made in Part 1 shouldn't show up. Have they been lost?*

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 main.

Edit the top of the `chkdict` *to add a line that prints your name after all of the* `import` *statements. Save the file when done.*

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.

`$` **{review and commit your change}**	Commit your change.
`$ git log --oneline`	**Summarize history for current branch.**
`$ git log --oneline main`	Summarize history for main branch.
`$ git checkout main`	Switch to main.
`$ ./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

Recall that main 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.

The git merge command automates the process of merging one branch into another. Here we'll merge your print-my-name branch into your main branch.

```
$ git checkout main                    Switch to main.
$ git merge print-my-name              Merge this branch into main.
```

The git merge command will prompt you to write a message for the merge commit. You can usually just use the default.

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.
```

This merge could happen automatically 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*.

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.

If a conflict arises, git leaves your working tree in a special funky state. The conflicting files have special markers inserted 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.

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

```
$ git branch conflict-demo origin/main    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.
```

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 `chkdict` in your code editor and search for these conflict markers.

Your mission to find *all* conflict markers and edit the files to somehow do what *both* branches intended.

Resolve the conflict in `chkdict`. Save your work.

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 an internal flag to realize that this commit represents a merge and not just a regular commit.

```
$ 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.
```

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.

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

8.1 Manipulating Branches

The git branch command provides tools for manipulating branches. You can do things like rename them, copy them, and so on.

A common workflow is to create a new branch for each feature that you work on — called a "feature branch." Once the feature branch is merged into the main branch, you can tidy things up by deleting it:

```
$ git branch -d print-my-name
```
Delete this branch.

You can use the `-D` mode if you're sure that it's safe to delete a branch.

The `-d` mode of git branch won't let you delete a branch unless git is confident that the commits of that branch are replicated elsewhere.

Part 3: Collaboration: GitHub

Topic 9: git remote, git fetch

Each git repository stores a list of other repositories that it knows about, known as *remotes*. You can download updates from remotes and *push* updates to them. 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 the git remote command.

```
$ git remote                    List named remotes.
$ git remote show origin        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.
```

Linking With Your GitHub Account

We'll now get your repository talking to your GitHub account. First, we need to *fork* — duplicate — the origin repository.

> *Navigate your browser to* https://github.com/pkgw/bloomdemo/*. Click the "Fork" button at the top-right of the screen. Once you're looking at your fork, click the "Clone or Download" button and copy the clone address to your computer's clipboard.*

The key thing about a fork is that you can do work on my code without needing to ask me for permission to start making changes. If you later want to submit your changes, the two repositories will nonetheless share a common git history so that they can be merged easily.

Now we'll tell your repo about your fork on GitHub, registering it as a remote named mine:

```
$ git remote add mine {GitHub clone address}    Register a new remote
$ git fetch mine                                Pull down its current status
```

Topic 10: git push

To share changes we *push* them to a remote. We'll create yet another branch with some quasi-nonsense changes:

```
$ git branch graduation origin/main             New branch with pristine files.
$ git checkout graduation                       Switch to conflict-demo branch.
$ mkdir alumni                                  Make a directory.
$ date >alumni/{your-github-username}.txt       Create your very own file.
$ git add alumni                                Register new directory with git.
$ git commit -m "Git graduation."               Commit
```

Of course, git push also uploads all of the needed backing data as well.

The command git push copies a branch from your local repository to a remote. The syntax with a colon shown below can be used to change the name that the branch will be given on the remote.

```
$ git branch -a                                 List all branches.
$ git push mine graduation                      Publish the changes.
$ git push mine main:experiments                Publish our main work as experiments.
$ git branch -a                                 List all branches.
$ git remote show mine                          Show details about mine.
```

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

You'll see that git prints some messages about creating a pull request. We'll ignore them for now.

Reload the GitHub webpage for your fork of the bloomdemo *repository. In the "Branch:" dropdown menu, you should see your new* graduation *branch. Click on the commit message ("Git graduation.") to see your change as displayed by GitHub.*

Your changes are now available to the wide world!

Issuing a Pull Request

If you're just working on a personal project, GitHub already adds value: above all else, it backs up your repository offsite. Where it really shines, however, is how it enables decentralized development. Specifically, GitHub *pull requests* (PRs) let you submit changes to repositories that you don't own.

Really, it would be more accurate to call them *merge requests*. GitLab.com is a service very similar to GitHub that does this.

Click the big green "Compare & pull request" button associated with your graduation *branch. You can attach a message to your pull request — in normal usage, this would summarize what changes are in your branch, and why.*

Once the first few pull requests arrive, the lab leader will demonstrate how they are reviewed and handled. In the meantime, you are encouraged to click around the different elements of the GitHub PR user interface!

Topic 11: git pull

There is a git pull command that somewhat mirrors git push. But really, git pull is just a combination of git fetch and git merge.

Wait for the instructor to merge a few pull requests into the main repository.

Now that your origin repository has seen some changes, let's incorporate them locally:

```
$ git checkout main          Switch to main branch.
$ git pull origin            Fetch and merge any changes from origin.
$ ls alumni                  See who's graduated!
```

It can be tricky to merge changes if your local tree is dirty. Before pulling, we recomend doing a git status and cleaning up any uncommitted or unstaged changes.

Grand Finale

Having successfully uploaded our work to the cloud, we can now shred our local 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 the main repository.
`$ cd bloomdemo`	Move into it.
`$ git remote add mine` **{GitHub clone address}**	Register our fork.
`$ git branch -a`	List all branches.
`$ git fetch mine`	Fetch our fork.
`$ git branch -a`	List all branches.
`$ git merge mine/experiments`	Restore our changes.
`$ ./chkdict` **{some words}**	Confirm our changes.

With a few short commands, you were able to recover 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`

Appendix: Unix Help

To learn about command-line programs in Unix operating systems, you can try reading their "manual page"s with the `man` command:

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

When you run `man`, you enter a special subprogram called the *pager*, described in the next Appendix. 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.

Appendix: The Pager, "less"

The *pager* is a special Unix program for navigating lengthy textual output. It offers many more features than the bare terminal. Because `man`, `git`, and many other Unix tools use it extensively, it is valuable 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`.

Pager programs come from a time before the scrollbar was even invented. When using `less`, you should **not** scroll your terminal window, because it won't interact well with `less`'s internal scrolling. Instead, navigate using the keyboard — `less` is controlled by commands that are mainly 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 of the file.
`>`	Go to the bottom of the file.
`/`	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.

The search and filter commands in `less` use a special formalism called *regular expressions* or *regexes* for matching text. In general, you can just type what you're looking for and the right thing will happen. *Many* Unix tools use regexes; they are just one of the many interlocking technologies that constitute the overall Unix ecosystem.

When in doubt, just press q to quit the pager and return to your main terminal prompt.

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
```
Page the file `README`.

Long-time Unix users might be used to paging files with the `more` command. The symmetry in the names is not accidental — `less` is `more`!

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.