Intro to git & GitHub

FELINE SABOTAGE PROTECTION FOR YOUR CODE

It's Saturday night. You've been working for two days on an extremely complex bit of code that returns an inverse square root.

```
float InvSqrt (float x){
    float xhalf = 0.5f*x;
    int i = *(int*)&x;
    i = 0x5f3759df - (i>>1);
    x = *(float*)&i;
    x = x*(1.5f - xhalf*x*x);
    return x;
}
```

Even with the four years of calculus you took, this one was a bear, but now — it works! You head off to meet with some friends, happy that a long week ended in such success.

Returning late that night, you push the cat off your pillow and fall asleep. In the morning, you wake up to your cat's meowing. You see her out of the corner of your eye *walking on your keyboard!* Frantic, you jump up to see your exquisite code has been replaced with this:

```
fg849**l    ts(( ==*````~<<
```

No, NO! This can't be! You hit CTRL-Z, then remember you're on a Mac and hit CMD-Z. More garbage. And then it sinks in: your code is lost forever. *You're doomed*.

CRASH! You wake up to see your cat has knocked over a mug on the desk. You jump up from the bed to see that the mug is intact. You rush over to your computer. Your code is fine! It was all a bad dream.

IT'S NOT REALLY CATS WE NEED TO WORRY ABOUT

Having perhaps unfairly maligned our feline friends, let me state that the biggest threat to your code is not outside agents but...you. Let's set aside the meandering cat scenario and look at a far more realistic one. You've been working on a large code base. Things are working — but your code base isn't complete.

You're launching into a new feature, one that will touch large portions of your existing, working code. You try things out, try more things, try alternatives — and after a while, your previously pristine code is in a precarious state. You've just changed so many things. And what used to work, doesn't anymore. This is enough to make the bravest developer falter.

Scenarios like this are exactly why git was created. Using a simple workflow with git (and GitHub), you'll have the confidence to work on code without fear that you'll completely hose it. Let's see how to work with it.

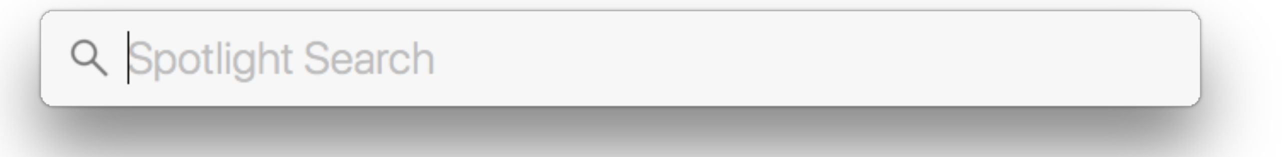
The workflow you use will depend on your exact situation. Are you starting from scratch — with no existing code base? Or are you working on, and contributing to, an existing application? Since the second scenario is the more common one, let's start from there. But first, you need to install git.

INSTALLING GIT

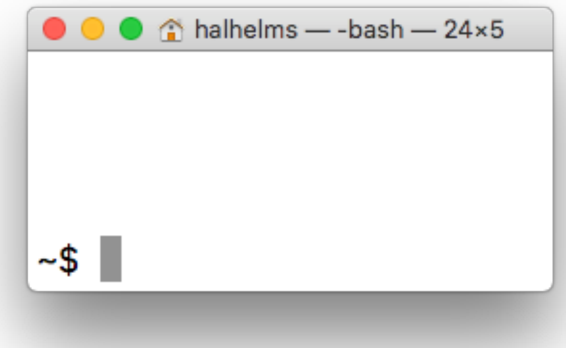
You can find a download for your operating system (Windows, Mac, Linux) at https://git-scm.com/downloads. Once you've downloaded it, run the installation program. Unless you have fairly extensive experience working with computers on a very technical level, you're unlikely to have needed to use your operating system's terminal before, but you will now.

The terminal has no modern graphical user interface (GUI), but is a relic of the days when all commands given to the computer were typed into a command line. Despite its ancient underpinnings, we suspect you'll come to accept its quirks and to prefer it over fancier interfaces for its succinctness. Let's use it now to make sure that you have git properly installed.

If you're working on a Mac, press CMD-spacebar to bring up Spotlight.

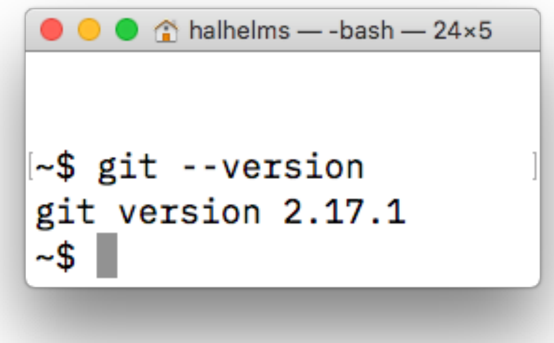


Begin typing `terminal`. Your Mac will fill in "Terminal.app". Press Enter and you'll get a first glimpse of your terminal app. It will look something like this (although yours may differ considerably):



```
~$
```

I know — not exactly welcoming, but give it a chance. Type `git --version` and press Enter. If git is installed on your system, the terminal should tell you the version number. (Don't be concerned with the actual number.)

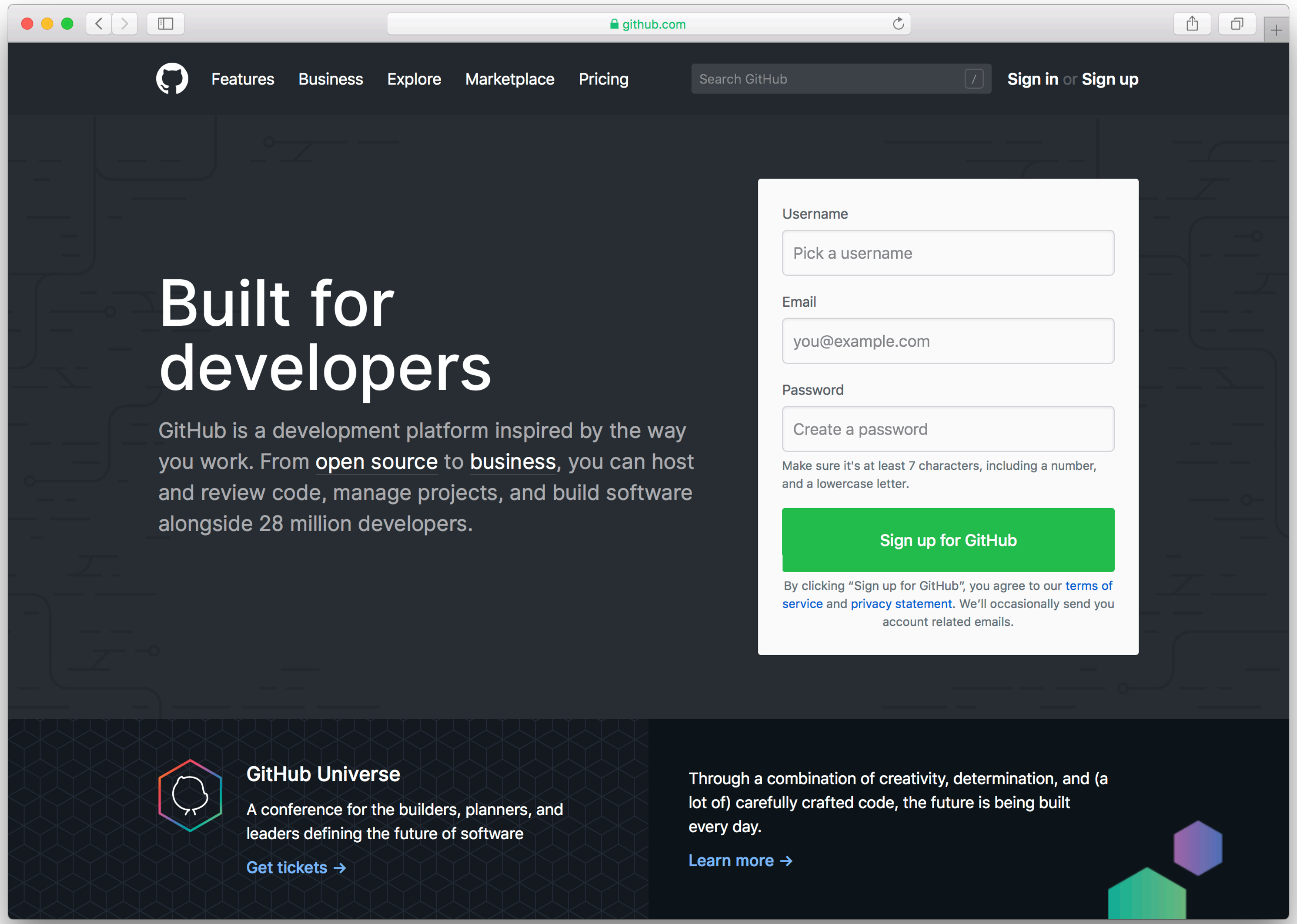


```
~$ git --version
git version 2.17.1
~$
```

WORKING WITH AN EXISTING CODE BASE

We're dealing with the scenario where you'll be working with an existing code base — called a *repository* (or more commonly, *repo*). In order to work with it, you'll need to get a copy of the code. This is known as *cloning a repo*.

For this Guide, you'll be cloning a WiseGuides repo from GitHub. GitHub is a website very commonly used by individual developers and companies to store their repos. In order to clone our repo, you'll need to set up a free GitHub account. Start by heading to https://github.com.



Built for developers

GitHub is a development platform inspired by the way you work. From open source to business, you can host and review code, manage projects, and build software alongside 28 million developers.

Make sure it's at least 7 characters, including a number, and a lowercase letter.

By clicking "Sign up for GitHub", you agree to our terms of service and privacy statement. We'll occasionally send you account related emails.

GitHub Universe

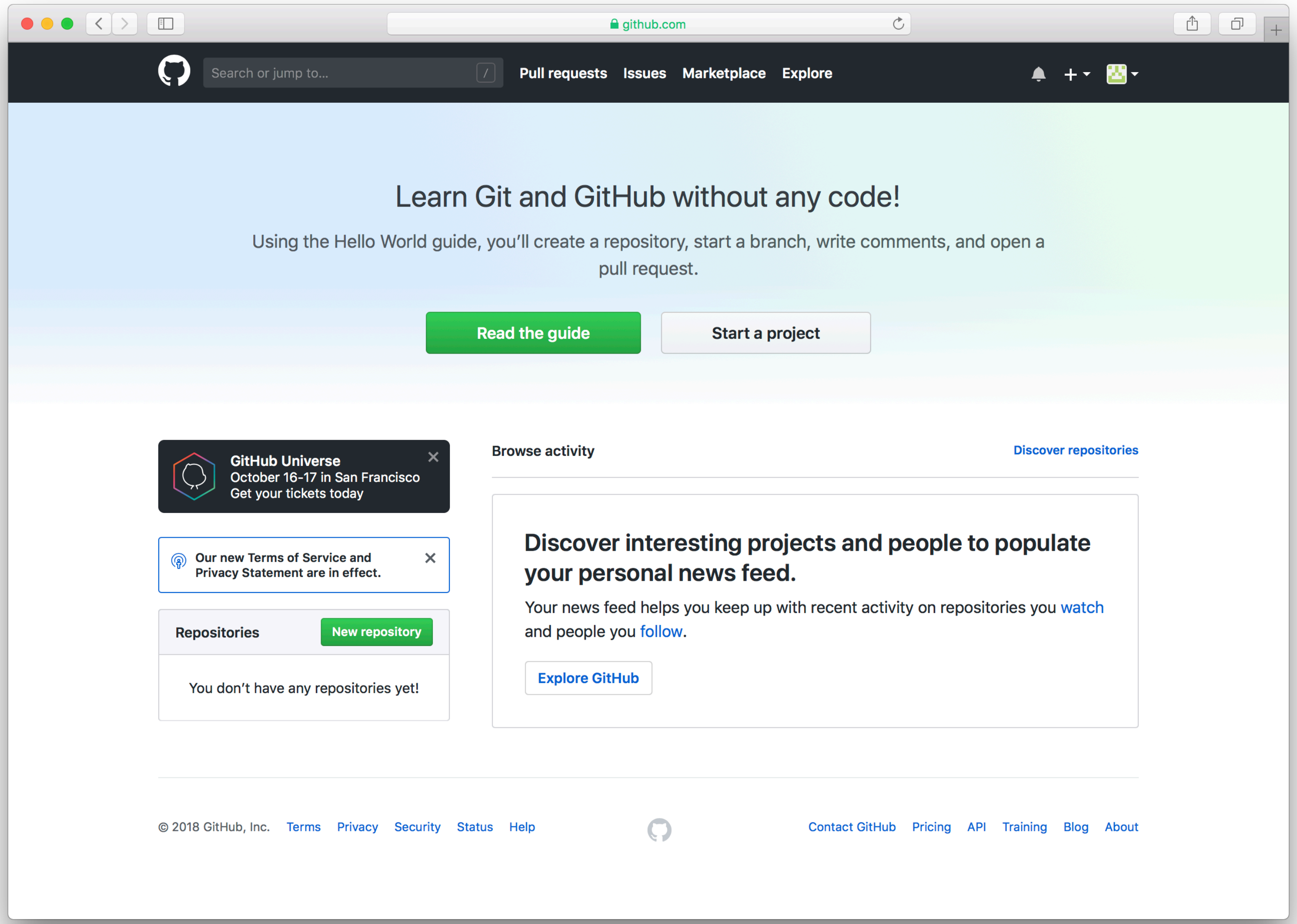
A conference for the builders, planners, and leaders defining the future of software

Through a combination of creativity, determination, and (a lot of) carefully crafted code, the future is being built every day.

Create an account. Follow the steps and select the Free account (it's selected by default) and continue.

You can skip the screen that asks you for your experience, *etc*.

When you're done, you should see this screen.



Learn Git and GitHub without any code!

Using the Hello World guide, you'll create a repository, start a branch, write comments, and open a pull request.

GitHub Universe
October 16-17 in San Francisco
Get your tickets today

Our new Terms of Service and Privacy Statement are in effect.

Repositories

You don't have any repositories yet!

Browse activity

Discover repositories

Discover interesting projects and people to populate your personal news feed.

Your news feed helps you keep up with recent activity on repositories you watch and people you follow.

© 2018 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub Pricing API Training Blog About

You're almost ready to clone your first repo. Let's do two things first.

Inside terminal, type: `git config --global user.name "Your Name"`

Replace "Your Name" with your name.

Press Enter then type: `git config --global user.email "you@domain.com"`

Replace "you@domain.com" with your email.

CLONING A REPO

Directories form a hierarchy of parents, children, grandchildren, etc. The "root" directory is the ultimate parent directory and, itself, has no parent.

Cloning a repo is quite simple. You begin by changing your directory to the *parent* directory that you want the repo to go into. In this case, you're going to start by *creating* that parent directory. You'll name it `wise-guides` and it will go directly under the `root` directory. (Over time, all these weird-sounding terms will become quite familiar).

To get to the `root` directory, type this into your terminal:

```
cd ~
```

Press Enter. From wherever you are in the directory hierarchy, this will always get you back to `root`.

From here, you're going to create that new `wise-guides` directory. Of course, you could do that with Mac's Finder program, but let's get you some practice working with the terminal. In terminal type:

```
mkdir wise-guides
```

Press Enter. You've just created a directory called `wise-guides`. To see all the contents of your root directory (including the newly-created `wise-guides`), type `ls` and press Enter. To get to the `wise-guides` directory, type `cd wise-guides` and press Enter. You can type `ls` and Enter again to see that there's nothing inside that directory. You're about to change that by cloning a repo available from the WiseGuides GitHub account.

Inside terminal type this (take your time to make sure you type it exactly correct):

```
git clone https://github.com/WiseGuides/intro-to-git.git
```

Press Enter. There should be a whir of activity in the terminal. When it stops, try entering `ls` again. You should see a new directory called `intro-to-git`. You can `cd` into that and you'll see that `intro-to-git` has some files in it.

Congratulations! You just cloned your first repo.

REMEMBER THE CAT?

We started out this Guide with a bad dream about a miscreant cat hosing your code, promising that, somehow, git was going to save you from your cat (and yourself). Let's see how.

Inside your newly-cloned directory is a file, `answer-to-life.html`. Open that file in VS Code. And there, in glorious HTML, is the answer to the question: What is the Meaning of Life? That's an important answer — what if your cat gets up to her old tricks? Not this time, Fluffy!

Your `wise-guides` directory is already "git-enabled". (All repos on GitHub are.) We're going to take advantage of that fact by using a git command. In terminal (and you should be in the `intro-to-git` directory) type:

```
git checkout -b bad-idea
```

Press Enter. You should see the feedback: `Switched to a new branch 'bad-idea'`.

Alright, let's delete that second paragraph — the one that begins with "What later versions of The Guide..." Go ahead: delete it. Probably not important anyway. Now, save the file. Close the file, then re-open it. That second paragraph? Gone.

Now you're thinking: *Maybe I should have read that paragraph before I deleted it.* Yeah, that was probably a `bad-idea`. Without the second paragraph, you've lost the meaning of life. Bummer.

Fluffy walks over to you, gives you a disgusted look, and types into the terminal.

```
git checkout master
```

When her paw hits Enter, your code is miraculously restored to its original version!

What exactly happened? What did Fluffy do?

Saying "git checkout" creates a "duplicate version" of the code isn't quite accurate, but you'll do fine if you think of it this way.

When you typed `git checkout -b bad-idea`, git created a duplicate version (called a *branch*) of your code and named it `bad-idea`. From that point on, you were making changes to the `bad-idea` branch.

When Fluffy checked out the `master` branch, she was checking out the original branch, where your deletions don't exist.

Let's do one other thing. Let's leave a note in the code to warn others *not* to delete the second paragraph. (Who knows? Next time Fluffy might not be around to save the day.)

But before we do anything, let's create a new branch. Let's call it `warning`.

```
git checkout -b warning
```

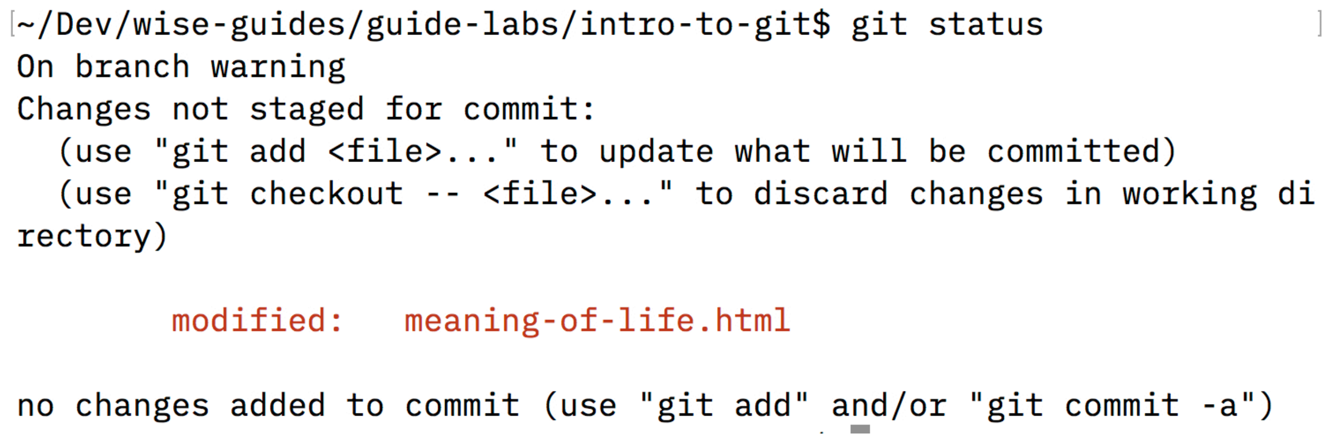
Press Enter and git will tell you you're in the new branch.

Now, add a third paragraph (wrapped in `<p></p>` tags) telling whoever comes along to avoid deleting anything.

OK — but that change only exists inside the `warning` branch. Somehow, we need to get it back into the `master` branch.

Type `git status` and press Enter. You should see something like this in your terminal:

Your terminal will look slightly different. That's OK.



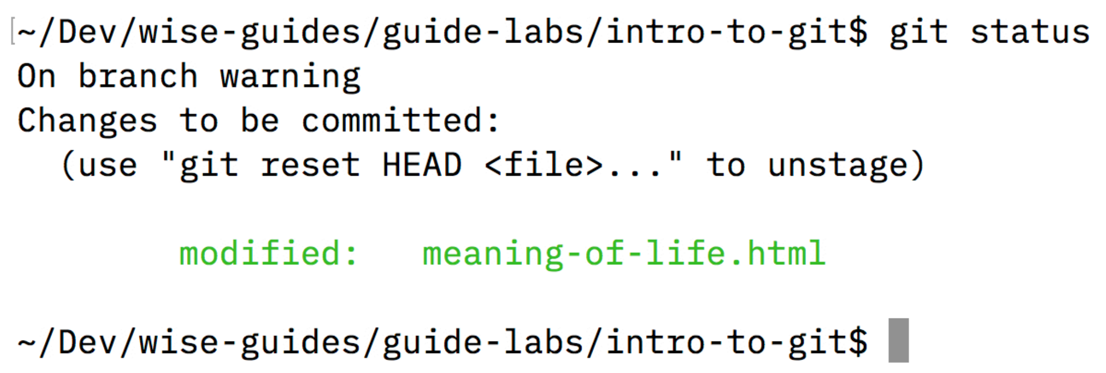
```
~/Dev/wise-guides/guide-labs/intro-to-git$ git status
On branch warning
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working di
rectory)

        modified:   meaning-of-life.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Say what? git is telling us that it knows we've made changes, but we haven't told git to make them part of the `warning` branch. This is good — we don't want every change we make to automatically become part of the branch we're working on. We'll add them explicitly. It's easy to add the changes in.

Inside terminal, type this:

```
git add -A
```

Press Enter and run `git status` again:

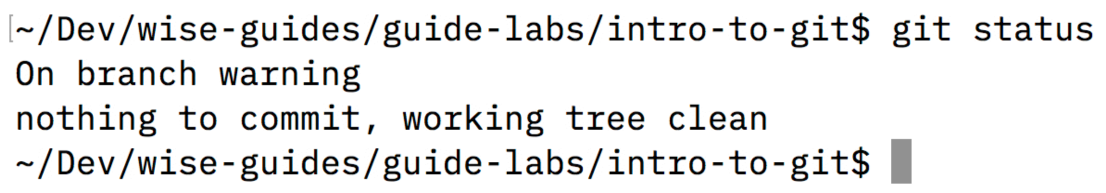


```
~/Dev/wise-guides/guide-labs/intro-to-git$ git status
On branch warning
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   meaning-of-life.html

~/Dev/wise-guides/guide-labs/intro-to-git$ 
```

This is our way of telling git "We want you to add these changes into our current branch." But we're not quite done yet. git wants to make sure we *really* want to add those changes. It's holding them in a special status called *staging* waiting for us to give the final word. Oh, that git — always so cautious. OK, let's commit the changes we just staged.

```
git commit -m "Added a warning for others not to delete anything"
```

Press Enter. Type `git status` again:



```
~/Dev/wise-guides/guide-labs/intro-to-git$ git status
On branch warning
nothing to commit, working tree clean
~/Dev/wise-guides/guide-labs/intro-to-git$ 
```

Nothing to commit, working tree clean? That sounds good! Now we can go back to our master branch.

```
git checkout master
```

Oh, no! The code we just wrote is gone. You look over to Fluffy. She's having none of it and strolls out of the room. Actually, everything is fine. You're back to your `master` branch — but you made the changes in the `warning` branch.

Let's go back to the warning branch.

```
git checkout warning
```

The code's back!

"*I don't need you, you traitor!*" you call to the retreating cat.

So, this is good: we have a branch that we did some work on. When we were ready to commit that work to the new branch, we added the files and committed it.

Now, when we checked out the original `master` branch, we have that version of the code. That's good! Two independent versions. But we do want to bring our changes into `master`. Also easy.

```
git checkout master
```

Now we're back in `master`, where our changes don't exist. So, let's merge `warning` into `master`.

```
git merge warning
```

And...the changes are merged.

PUTTING IT INTO PRACTICE

Now it's your turn to try out what you've learned with our `Intro to git & GitHub: Lab Guide`