CS312 - Git Lab
Spring 2012

David Kauchak

Git is a decentralized, version control system that allows you to manage your own individual projects, however, the real power comes when you start collaborating on a project with other users. There is lots of information about Git online. I've listed some resources on the course web page, but feel free to look around at others as well.

Today, we're going to walk through some of the different features of Git. To start with you can work on your own, but for the last part of this lab, you'll need to work with others in the class.

Git basics

Creating an initial project to play with

Open up a terminal window and `cd` into a directory where you can play around in.

Create a new directory called `myproject` and the `cd` into that directory.

Git allows us to keep track of changes to a project and it's files, so to get started, create a file called `simple.rb` with your favorite text editor and copy and paste the following code into it:

```
class Simple
     attr_accessor :data

     def initialize(data)
         @data = data
     end
end
```

and save it into the `myproject` directory.

Git setup

Git is a command-line utility. To get started, let's setup some global information that git will use. You only need to do this once (unless you use Git on your laptop).

```
$ git config --global user.name "Your Name Comes Here"
$ git config --global user.email you@yourdomain.example.com
```

Creating a Git repository

Now that we're all setup and we'd like to start tracking our files. To do this, we need to create a new Git repository.

```
$ git init
```

Creates a new empty repository. If you're curious, Git creates a directory `.git` and stores all the information in there. You're welcome to poke around in there, but when you're done, `cd` back into the `myproject` directory.

Now, we'd like to tell Git that we want to add our `simpe.rb` file to the repository:

```
$ git add simple.rb
```

and now let's commit these changes to the repository so that our repository isn't empty.

```
$ git commit
```

Anytime you commit files (whether it's creating a new project or latter on) you will be prompted in your default text editor[1] to enter a short message about this commit. The general rule of thumb is to give a short sentence (< 80 characters) summarizing what you're committing. If you need more space than this to describe your commits. Put a summary at the top, followed by a blank line, followed by the more detailed description.

Save this file and exit the editor.[2]

You will get some message like the following if it works properly:

```
[master (root-commit) 3b5cf42] Creating a new project
 1 files changed, 6 insertions(+), 0 deletions(-)
 create mode 100644 simple.rb
```

that summarizes what you're committing. In this case, just 1 changed file with 6 lines inserted.

[1] If you don't like your default text editor type `export EDITOR=some_editor` and this will be used as your default editor instead. If you'd like this to be permanent, put this command in your `~/.bashrc` file.

[2] If you don't like the whole editor thing, you can use the `-m` flag to include a message with your commit call `git commit -m "This is my summary of this commit"`

Git fun

Now we have our basic project under version control. At any point we can ask Git what's going on with our current project by typing:

```
$ git status
```

Since we haven't done anything yet, it should just tell you that we're working on the master branch (the main branch– more about branches soon) and that no changes have been made in our current working directory.

```
# On branch master
nothing to commit (working directory clean)
```

Now we can start adding to the project. Let's add another file in the project that tests the functionality of our basic class. Create a file called `test_simple.rb` with the following contents:

```
require_relative "simple"

s = Simple.new("some data")
puts "Before: " + s.data
s.data = "some different data"
puts "After: " + s.data
```

Run this file through Ruby and make sure it's working. If we again ask Git what the status is we'll get the following message:

```
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# test_simple.rb
```

indicating that there are some files that are "untracked" that is haven't been added to Git.

Add this file to Git.

```
$ git add test_simple.rb
```

The `add` command adds any changes we've made to the current *index*, but it does **NOT** commit them to the repository yet. It just tells Git that next time we want to commit to the repository, include any changes to this file in the commit. You can see this by checking the status again:

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   test_simple.rb
#
```

Before we commit these changes, we can ask Git what is the difference between what is in the index (i.e. what we're ready to commit) and what is currently in the repository using `diff`:

```
$ git diff --cached
```

The `--cached` tells Git to compare the index to the repository (more on `diff` without `--cached` in a bit). The message we get back:

```
diff --git a/test_simple.rb b/test_simple.rb
new file mode 100644
index 0000000..9e55669
--- /dev/null
+++ b/test_simple.rb
@@ -0,0 +1,6 @@
+require_relative "simple"
+
+s = Simple.new("some data")
+puts "Before: " + s.data
+s.data = "some different data"
+puts "After: " + s.data
```

tells us that `test_simple.rb` is a new file and has 6 new inserted lines (denoted by the '+').

Let's go ahead and commit these changes.

```
$ git commit
```

More commits

Now let's make some more changes to our data. Add a `to_s` method to `simple.rb`:

```
  def to_s
    "Simple: #{@data}"
  end
```

and add some code in `test_simple.rb` to test this new function:

```
puts s
```

If we now ask Git for the status we see that we have two "modified" files:

```
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified:   simple.rb
# modified:   test_simple.rb
#
```

We can again use `diff` to see what those differences are. If you used the `--cached` version:

```
$ git diff --cached
```

We *don't* get any differences. This is because we have **not added any of these changes to the index**. If we call `diff` without `--cached` we will see these differences.

```
$ git diff
```

Shows us:

```
diff --git a/simple.rb b/simple.rb
index f6cbf01..0a01610 100644
--- a/simple.rb
+++ b/simple.rb
@@ -1,6 +1,11 @@
 class Simple
        attr_accessor :data

-       def initialize(data) @data = data
+       def initialize(data)
+               @data = data
+       end
+
+       def to_s
+               "Simple: #{@data}"
        end
 end
diff --git a/test_simple.rb b/test_simple.rb
index 9e55669..5412a5f 100644
```

```
--- a/test_simple.rb
+++ b/test_simple.rb
@@ -4,3 +4,5 @@ s = Simple.new("some data")
 puts "Before: " + s.data
 s.data = "some different data"
 puts "After: " + s.data
+
+puts s
```

Notice now that we now see differences in the two different files. When we call `diff` without `--cached` it shows us differences between the repository and any files that are in the repository that are not currently in the index (i.e have been added).

There are two ways that you can interact with the Git repository. (Use one of these approaches to commit the changes above.)

1. If you just want Git to track and commit *all* changes to any files in the repository, then whenever you create a new file you add it using `git add`. When you want to commit changes you can use:

   ```
   $ git commit -a
   ```

   which tells Git to commit all differences in files (`-a` stands for all). This should feel similar to other version control systems you may have used.

2. If you want finer grained control then you can use `git add` in combination with `git commit` (without the `-a`). In this situation, the `add` command adds any changes from a specified file to the *index* and then when you type `git commit` it only commits the changes in the index to the repository. Notice that with this paradigm you must use add to each time you change a file. This is generally the preferred approach since it allows you to specifically control which files you're committing to the repository.

   To make it clear, if we were to submit our changes above we'd type:

   ```
   $ git add simple.rb  test_simple.rb
   $ git commit
   ```

   You could do two separate add commands or one like I did above.

Logging

So far you may be saying, ok, great, but what does this buy me? Even in the use case above, each commit creates a snapshot of your code over time and is stored in the repository. You can see all of these changes by typing:

```
$ git log
```

which will output a high-level log of the commits with the most recent on top

```
commit 8c07fca169d0d829e58441b5687e45f08847c3be
Author: David Kauchak <dkauchak@middlebury.edu>
Date:   Wed Feb 22 16:03:53 2012 -0500

    Added the to_s method and test cases

commit c8d40e981ade7add6cfb3142d88ef41fd6f3d28e
Author: David Kauchak <dkauchak@middlebury.edu>
Date:   Wed Feb 22 15:26:28 2012 -0500

    Added a testing file

commit 3b5cf4219a80c0f018afd6f2b621e771f3469de3
Author: David Kauchak <dkauchak@middlebury.edu>
Date:   Wed Feb 22 15:11:45 2012 -0500

    Creating a new project
```

You can see here why it's important that you register your name and e-mail and how the messages play a roll. If you want more detail you can type:

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

and you'll get a bit more information

```
commit 8c07fca169d0d829e58441b5687e45f08847c3be
Author: David Kauchak <dkauchak@middlebury.edu>
Date:   Wed Feb 22 16:03:53 2012 -0500

    Added the to_s method and test cases

 simple.rb      |    7 ++++++-
 test_simple.rb |    2 ++
 2 files changed, 8 insertions(+), 1 deletions(-)

commit c8d40e981ade7add6cfb3142d88ef41fd6f3d28e
Author: David Kauchak <dkauchak@middlebury.edu>
Date:   Wed Feb 22 15:26:28 2012 -0500
```

```
    Added a testing file

 test_simple.rb |    6 ++++++
 1 files changed, 6 insertions(+), 0 deletions(-)
 create mode 100644 test_simple.rb

commit 3b5cf4219a80c0f018afd6f2b621e771f3469de3
Author: David Kauchak <dkauchak@middlebury.edu>
Date:   Wed Feb 22 15:11:45 2012 -0500

    Creating a new project

 simple.rb |    6 ++++++
 1 files changed, 6 insertions(+), 0 deletions(-)
 create mode 100644 simple.rb
```

You can see all of the commits, which files were changes and what the changes were (+'s signify line additions and -'s deletions). `git log` has lots of other options you can play with, for example if you want lots of detail:

```
$ git log -p
```

If you want to drill down and see the changes between any different versions (for example to undo some of them), you can use diff with the commit number (that long number after "committ"). For example, from my logs above I could type:

```
$ git diff 3b5cf4219a80c0f018afd6f2b621e771f3469de3
```

and that would give me the difference between my current files and the commit number specified (it will be a different number for you). If you want to compare two different commits to each other, you can put both of them on the line.

If you want to just look at a particular file (or set of files) you can also include these in the diff command following the commit number, separated by a `--`:

```
$ git diff 3b5cf4219a80c0f018afd6f2b621e771f3469de3 -- test_simple.rb
```

Another way to view the past history is with the `show` command. You can give any commit number and it will show you what the changes were:

```
$ git show c8d40e981ade7add6cfb3142d88ef41fd6f3d28e
```

Making major changes: branching

With the `diff` commands above you can look at individual file differences and could undo changes by seeing what was changed from an older version, making these edits again and then checking them back in. This is fine for minor changes, however, sometimes you know that you're going to be making some major changes to the code and you're not sure if they're going to work. Git allows you to create a *branch* of the main trunk of commits (called the *master*) and you can commit to that branch as many times as you want. If you're satisfied that your branch worked you can *merge* it back in with the main trunk. If, however, it doesn't work out, you can just discard the branch and go back to the trunk.

Working by yourself, this is a pretty nice feature. When you're working with others, this is **very** important since it 1) allows the main project to continue to move forward while 2) still allowing you to commit your work into Git.

To create a new branch you use the `branch` command:

```
$ git branch newfeature
```

where `newfeature` is the name of the branch.

We can see what branches are available for this repository with the `branch` command without any arguments

```
$ git branch
```

which gives us the two branches:

```
* master
  newfeature
```

The '*' indicates which branch we're currently viewing.

Now that we've created a new branch, we can switch to it by using the `checkout` command:

```
$ git checkout newfeature
```

Right now, our files should look the same, however, we can edit the files and commit to this new branch just like we could before without changing the original master branch.

Let's say that we want to allow the constructor of our `Simple` class to be called without any parameters and so we change `simple.rb` to have:

```
def initialize(data = 0)
```

and then add a test in `test_simple.rb` at the end:

```
s2 = Simple.new()
puts s2
```

Commit these changes to the repository using commit.

Now the master branch and the newfeature branch contain different contents. You can see this difference using `diff`:

```
$ git diff  master newfeature
```

Another cool thing you can do, though, is switch back and forth between branches.

```
$ git checkout master
```

switches back to the master branch.

```
Switched to branch 'newfeature'
```

If you type

```
$ git branch
```

you'll see the '*' has changed back to the master branch. If you look at your files, you'll also notice that amazingly they've changed back to the master branch's version (you can switch back and forth if you want and watch the files change). If you'd created or deleted any files, these would also switch back and forth as you switched branches.

After working on the branch, hopefully, you discover that this new feature is a good thing and you finish coding the feature. At this point, what you'd like to do is incorporate your changes back into the master branch. If you're working on a collaborative project, this master branch may have also changed along the way, but in our case it hasn't. In either case, you still need to *merge* your branch with the main branch.

If you haven't already, switch to the master branch.

Now, we can merge the newfeature branch into the master branch with the `merge` command:

```
$ git merge newfeature
```

In this case, we Git was able to automatically update the files:

```
Updating 8c07fca..7e93369
Fast-forward
 simple.rb      |    2 +-
 test_simple.rb |    3 +++
 2 files changed, 4 insertions(+), 1 deletions(-)
```

This occurred because the master branch had not been modified.

Let's create a conflict so that you can see what happens when you try and merge a branch with a conflict.

Check again to make sure that you're in the master branch (if now, switch using `git checkout master`). Edit the `simple.rb` file and change the `to_s` method to

```
"Data: #{@data}"
```

Commit these changes.

Now switch to the newfeature branch and edit the `simple.rb` file to be:

```
"Simple-data: #{@data}"
```

and commit these changes.

Switch back to the master and then try and merge the newfeature branch in:

```
$ git merge newfeature
```

In this case, Git cannot automatically merge the files together so it gives you a warning:

```
Auto-merging simple.rb
CONFLICT (content): Merge conflict in simple.rb
Automatic merge failed; fix conflicts and then commit the result.
```

In addition to the warning, it will also annotate the files that need to be merged. You can see which ones these are with `diff`:

```
$ git diff
```

Git has highlighted the differences in `simple.rb`.

```
diff --cc simple.rb
index 5e70a1a,566696a..0000000
```

```
--- a/simple.rb
+++ b/simple.rb
@@@ -6,6 -6,6 +6,10 @@@ class Simpl
         end

         def to_s
++<<<<<<< HEAD
 +               "Data: #{@data}"
++=======
+                "Simple-data: #{@data}"
++>>>>>>> newfeature
         end
   end
```

Open `simple.rb` and you'll see the merged annotations. Go ahead and fix it so that it stays with the original master's version and then commit the changes.

Eventually, you'll reach a point where you no longer need a branch. This can happen for two reasons:

1. You merged your results in. In this case, you can delete the branch as follows:

   ```
   $ git branch -d newfeature
   ```

   The `-d` ensures that you've actually merged into the current branch before it allows you to delete it.

2. Sometimes you decide that a branch was a crazy idea and you never want to see it again. In this case you won't be merging in the branch with the master branch and just want to delete it:

   ```
   $ git branch -D newfeature
   ```

   Be careful, though. Once you delete a branch it's gone.

GitHub

Everything we've done so far has been mostly for an individual project. In these situations, you will often have a general repository that everyone uses (though someone must manage it). If you still have some time left after completing the above (some time being 15 minutes or so), try and do the following, which will walk you through the basic steps for collaboration.

This exercise will work better if you get into a group of 3-4 people. Only one person needs to type (i.e. create an account, etc.). The other people in the group can make sure that she/he is doing the right thing :)

For not, we're going to use github, though eventually, I will setup a git server internally that we can use.

1. Sign-up for an account on https://github.com/

   (a) Click on the button that says "Plans, Pricing and Signup"

   (b) Click on the button that says "Create a free account"

   (c) Enter the relevant information and click continue.

2. github (along with many git servers) uses ssh to connect to it. Because of this, you'll need to setup an ssh key. Go to:

   http://help.github.com/linux-set-up-git/

   and follow the instructions in the section "Next: Set Up SSH Keys"

   Stop after the `Test everyting out` section.

3. Now that you're all setup on github you can track down the demo class project and get your own copy.

   (a) Go to:
   https://github.com/dkauchak/cs312-middlebury-tutorial/
   which is the repository I setup.

   (b) Since it's a public repository (that I'm the owner of) you can just muck around in it. Instead, what you do is create your own copy of the repository. In github, before you can grab your own copy you need to *fork* off your own copy (this is similar to branching). Click on the "Fork" button from this web page to create your own copy.

   (c) Now, we can *clone* this repository to create our own version. In terminal, type:

   ```
   $ git clone git@github.com:username/cs312-middlebury-tutorial.git
   ```

   where `username` is your github username. This step clones or copies the git repository into your local directory. Now you have a working git repository that you can do all the things we did on above.
   When a repository is cloned, it has a default remote called *origin* that points to your fork on GitHub, not the original repo it was forked from. To keep track of the original repo, you need to add another remote, we'll call it upstream:

   ```
   $ cd cs312-middlebury-tutorial
   $ git remote add upstream git://github.com/dkauchak/cs312-middlebury-tutorial.git
   $ git fetch upstream
   ```

   Think of this like creating another branch. The 2nd line links the name *upstream* to the original github repository and then the *fetch* command makes sure that you have the latest version.

4. Now that you have you're own working version you can make some changes to your local repository.

(a) Go into the `data` directory and edit the three files with information about people in your group. Since this will technically be public, just use your last initial.

(b) Once you've made all the changes that you want, commit them to your local repository just like you in the tutorial above.

5. Now you're local repository is out of synch with your remote fork on github. You can push the changes out to your fork with the following command:

   ```
   $ git push origin master
   ```

   Recall that by default your forked remote repository is called `origin`. This command will push your local master branch to your github repository.

   If you now go back to github and look at your local fork (not the original one for username dkauchak), you should see your changes in the remote repository.

6. Eventually, you'd like to see your changes reflected in the main repository (i.e. the one owned by me that you forked from). To do this, you institute a "pull request" which asks me (the owner of the main repository) to incorporate your changes.

   There is a button on your github project in the upper right that says "Pull request". Click this button and fill out the relevant information. This will send me an e-mail requesting the integration.

   Come talk to me and I'll merge it for you :)

7. Once I've merged it into the main branch, you can update your local repository again from the main branch and see your changes (along with other changes that other people have made):

   ```
   $ git fetch upstream
   $ git merge upstream/master
   ```

   The first command tells git to grab the latest version from upstream (my original repository on github) and then the merge command merges it with your local version. upstream/master specifies the remote location (upstream) and the branch (master).

A few notes on GitHub

GitHub is an online interface for Git project management. All of the commands above can be done without Git using your own Git server as long as others can view it. We'll play some more with this when you all start working on your final project.

See the course web page for more tutorials.