lecture4

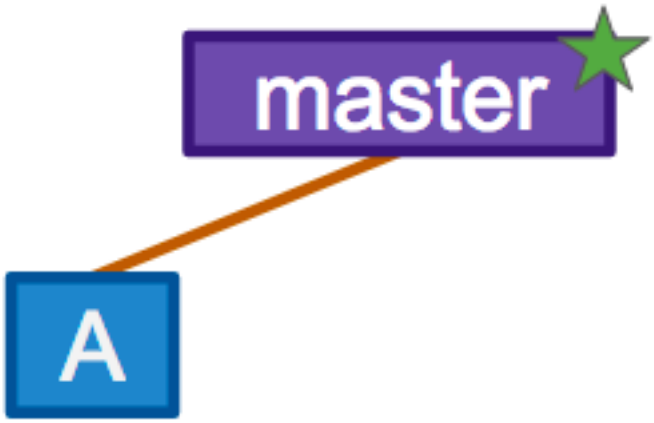
September 18, 2020

1 Lecture 4: September 17, 2019

1.1 Branches in `Git`

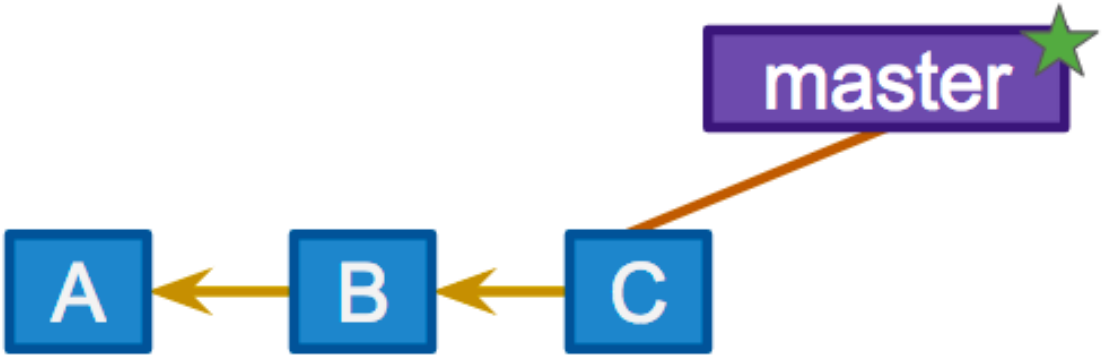
1.2 Basic Branching

- We have encountered branches a few times so far but we haven't really said much about what they are or why they're important. They are very important. In fact, they form a core piece of the `git` workflow.
- A `git` branch is a "Sticky Note" on the graph. When you switch branches you are moving the "Sticky Note".
- Suppose you have a newly initialized repository. Your first commit is represented by the **A** block in the figure below.



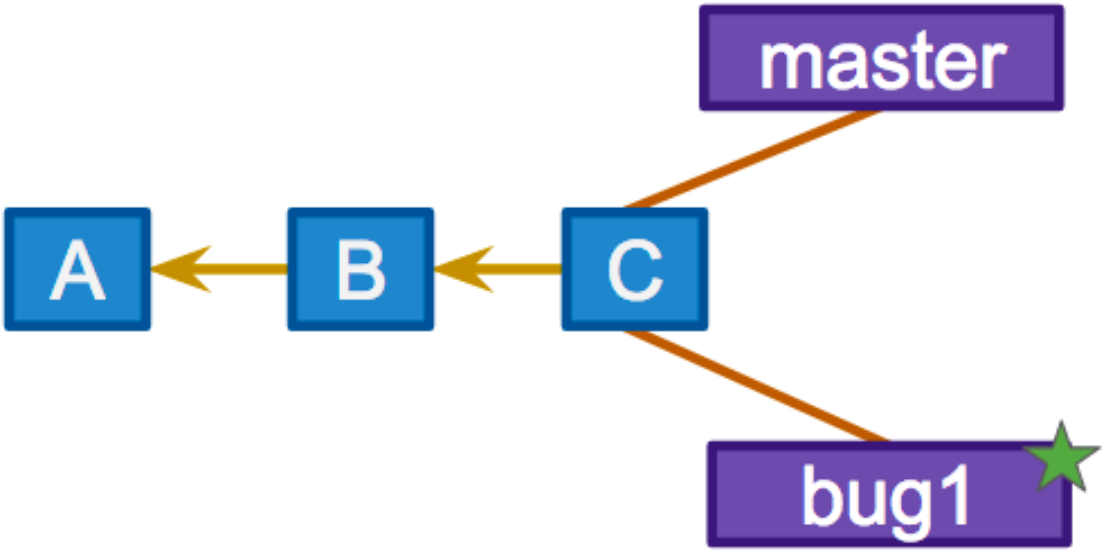
A default branch is created and `git` named it `master`. The name `master` has no special meaning to `git`.

Now suppose we make a set of two commits (B and C). The `master` branch (and our pointer) moves



along.

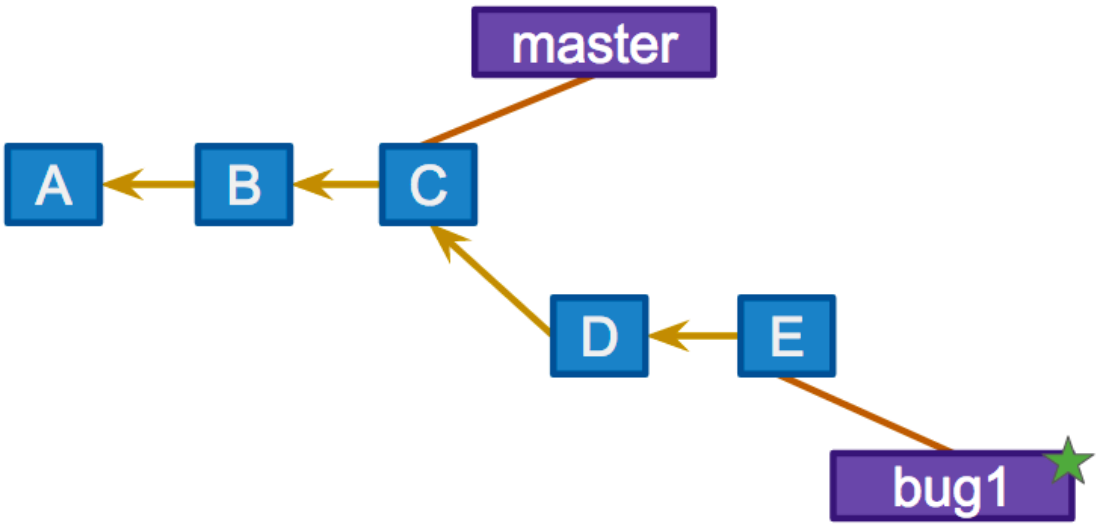
So far so good. Suddenly we find a bug! We could work on the bug in `master` but that's not really a good idea. It would make a lot more sense to `branch` off of `master` and fix the bug on its own `branch`. That way, we don't interfere with things on `master`. We'll discuss the details of how to create branches in the lecture exercises. For now, suppose we create a new branch called



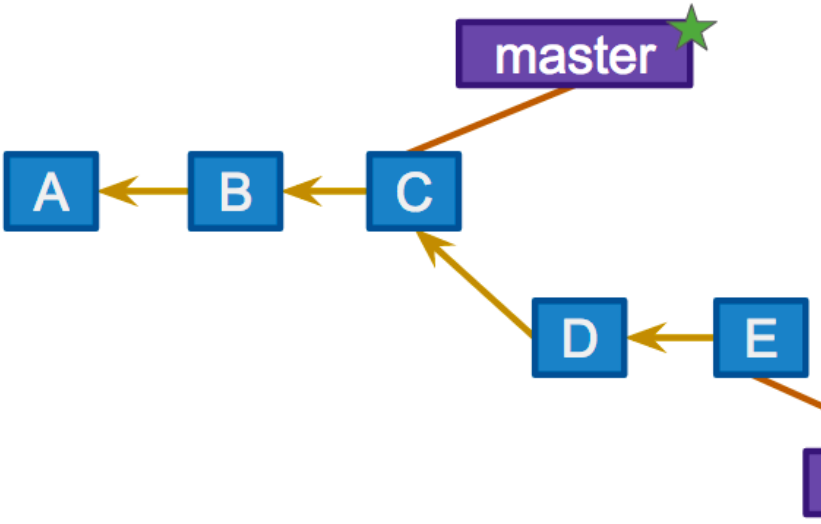
`bug1`.

The new branch is a pointer to the same commit as the `master` branch (commit **C**) but the pointer moved from the `master` branch to the `bug1` branch.

We do some work on the `bug1` branch and make two more commits. The pointer and branch now move to commit **E**.



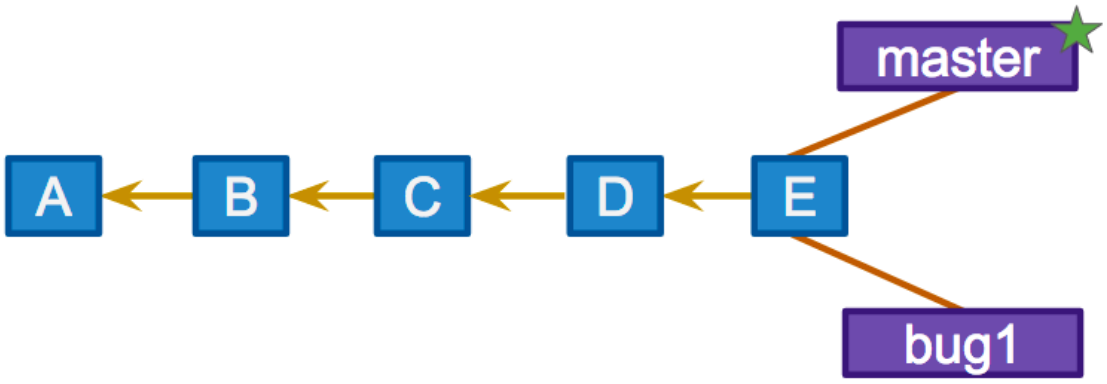
- Now you decide that the bug you found has been fixed.
- You've modified a file and maybe even added a new file.



- You can switch back to the `master` branch.
- What you'll see is that none of the files that you just fixed and/or created are in your working directory!

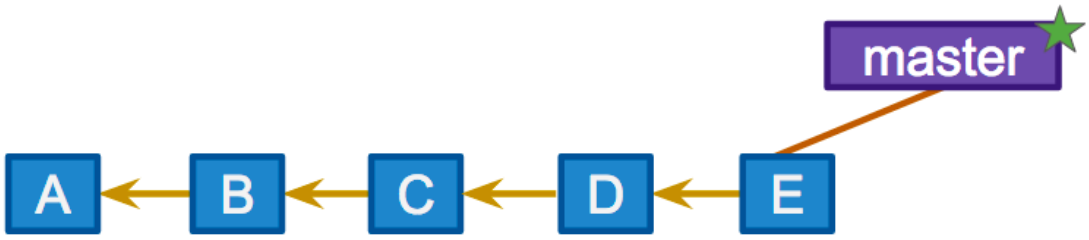
- The first couple of times you see this, it feels really uncomfortable.

- However, this is the correct `git` workflow: `git` works with snapshots!
- How do we get the bug fix into our `master` branch?
- We already know the command. From the `master` branch, just do `git merge bug1`.



This looks really nice! The `merge` brought the two change histories together prefectly.

- The only thing left to do is to delete the `bug1` branch.
- We don't need it anymore and so we really don't want it floating around.
- To delete a branch you simply write `git branch -d bug1`.

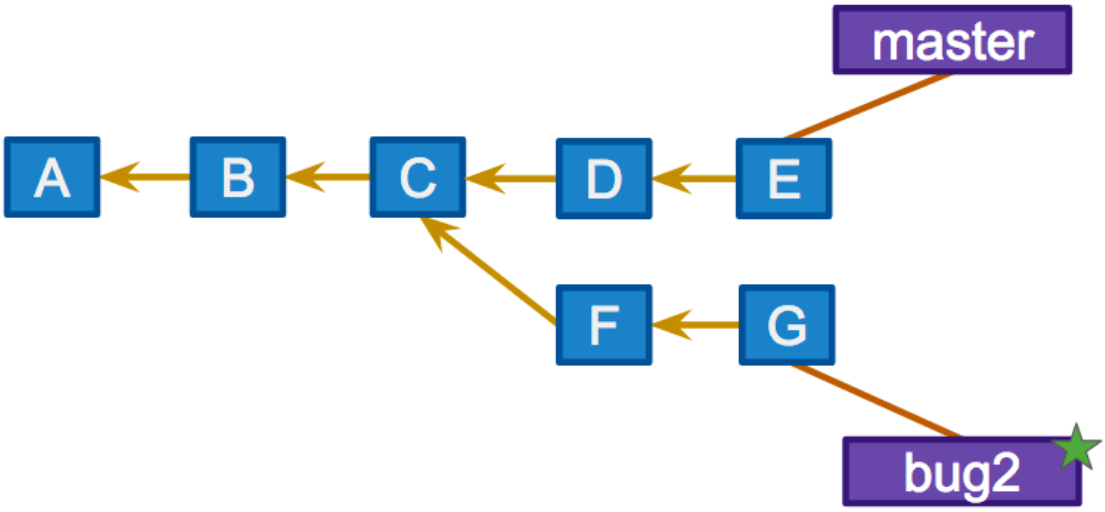


This looks like a nice clean tree now. If only things were always this simple.

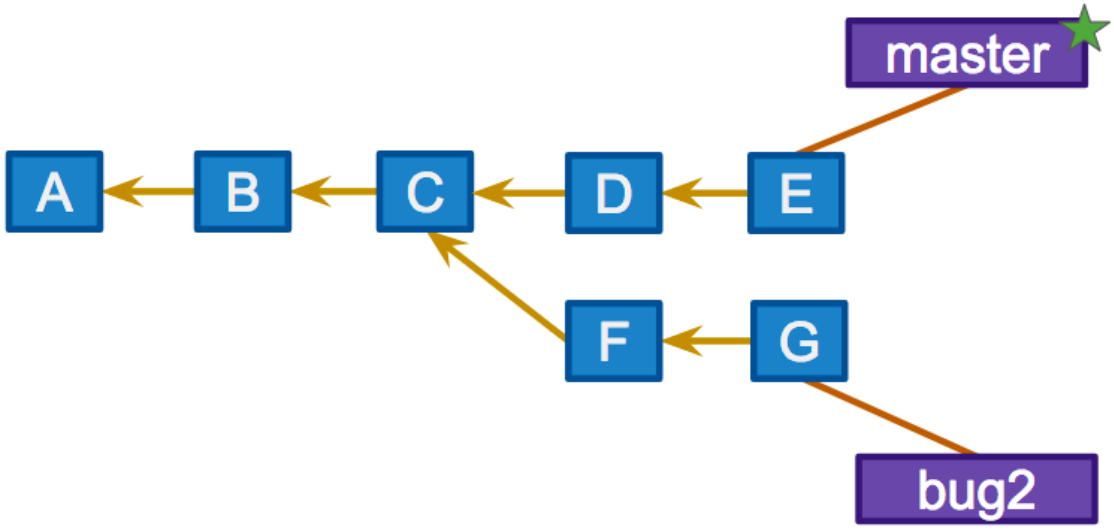
1.3 Nonlinear Histories: Workflow Choices

Another common scenario is as follows: * We created our "story branch" off of commit **C** to address some bug (not the same bug as before!). Call this branch `bug2`. * However, some changes have happened in `master` since we branched off of **C**. For example, `bug1` has been merged into `master`. * We have made a couple of commits on `bug2`.

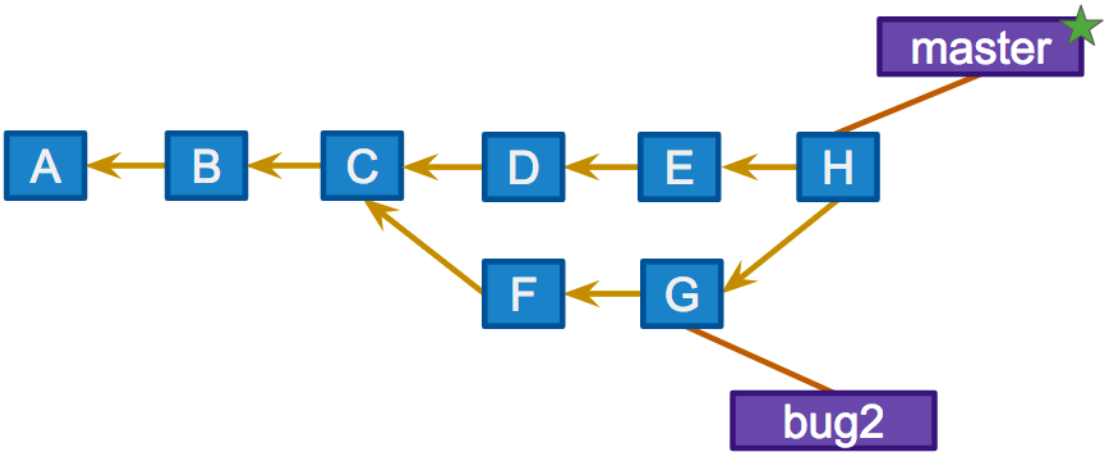
Here's the current graph:



Once we're ready to merge our bug fix, we switch back to `master`.

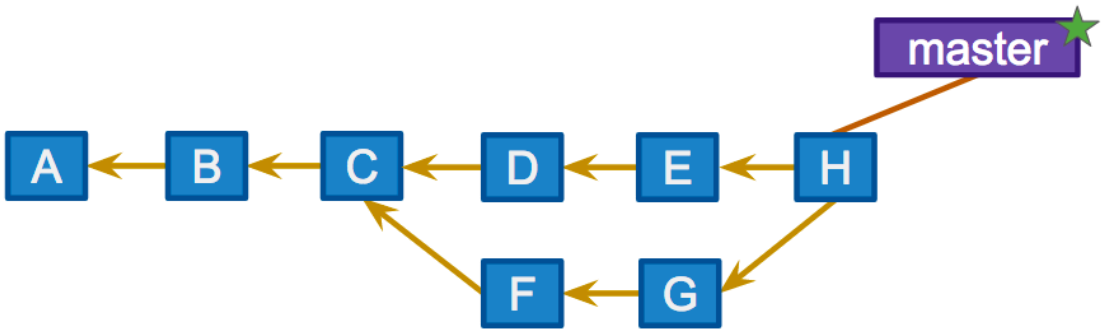


Now we attempt to merge. Our attempted merge should connect the new version H to both E and G (H came from E and G).



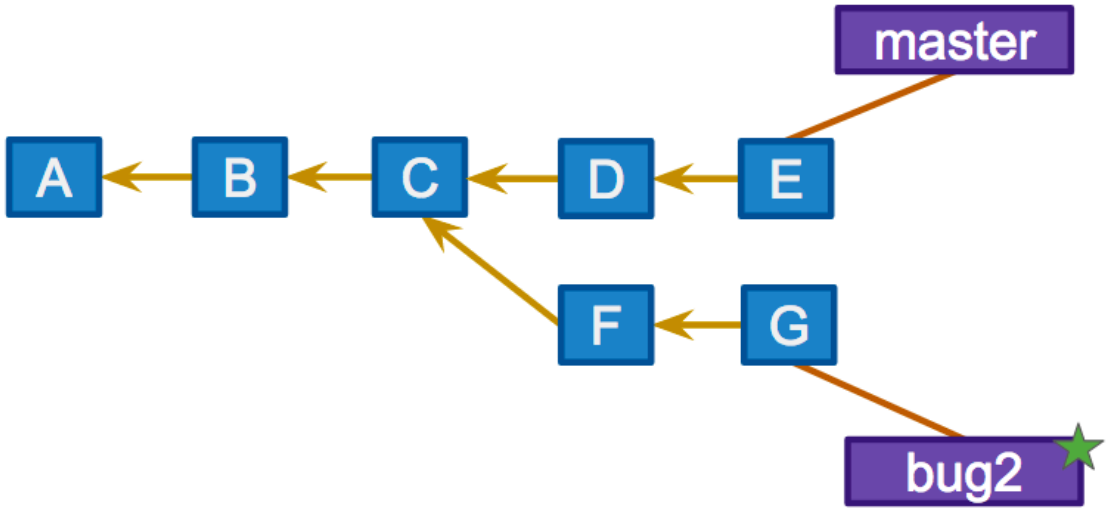
This merge can be quite unpleasant if there are conflicts.

Now we delete the `bug2` branch since the bug fix has been successfully merged into `master`.

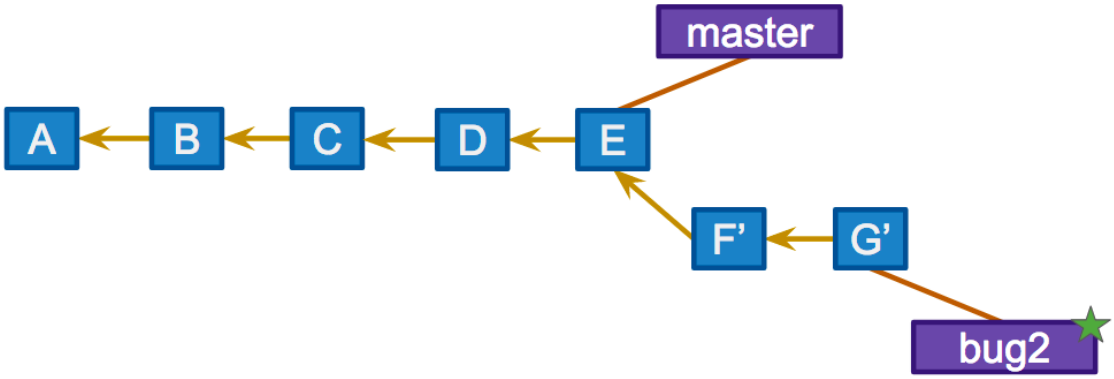


- The graph is now a bit of a mess; the history is nonlinear.
- There's nothing particularly wrong about this.
- However, such a history makes it hard to see the changes independently.
- What if another branch came off of **G**? You could have multiple loops!

There is another way to do merges that helps "linearize" the graph. Let's pick up with our `bug2` branch just before we switched to the `master` branch for a merge.



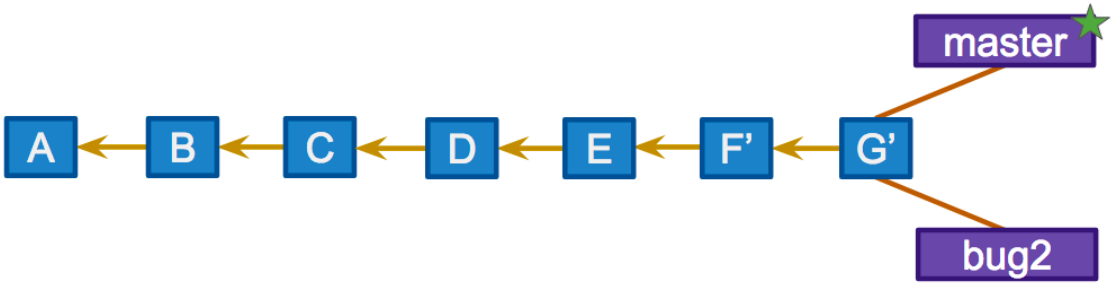
This time, instead of starting the `merge` process right away, we'll first `rebase`.



`git rebase master`.

What does `rebase` do? * Undo the changes we made off of **C**, but remember what they were * Re-apply those changes on **E** instead

Now we proceed as usual: * `git checkout master` * `git merge bug2`



- Now we get a nice linear flow.

Comments on `rebase`

- The actual change set ordering in the repo mirrors what actually happened. That is, **F'** and **G'** came after **E** rather than in parallel to it.
- We have re-written history; this is controversial.

Basic Rule: Don't rebase public history Never rebase commits once they've been pushed to a public repository.

Some Rough `rebase` guidelines Use an interactive rebase to polish a feature branch before merging it into the main code base.

1.4 End of Slideshow — Begin lecture exercises

In Lecture 2, you worked with the playground repository. You learned how to navigate the repository from the `Git` point of view, make changes to the repo, and work with the remote repo.

One very important topic in `Git` involves the concept of the branch. You will work **extensively** with branches in any real project. In fact, branches are central to the `Git` workflow. In this portion of the lecture, we will discuss branches with `Git`.

For more details on branches in `Git` see Chapter 3 of the `Git` Book: Git Branching - Branches in a Nutshell.

1.5 Branching

As you might have noticed by now, everything in `Git` is a branch. We have branches on remote (upstream) repositories, copies of remote branches in our local repository, and branches on local repositories which (so far) track remote branches (or more precisely local copies of remote repositories).

Begin today's lecture by entering your **course repository**. Note that the following cell is not necessary for you. I have to re-clone the repo since I'm in a new notebook. You should just keep working like usual from the command line.

[3]:
```
%%bash
cd /tmp
rm -rf cs207_david_sondak #remove if it exists
git clone https://github.com/dsondak/cs207_david_sondak.git
```

```
Cloning into 'cs207_david_sondak'…
```

Once you're in your course repo, you can look at all the branches and print out a lot of information to the screen.

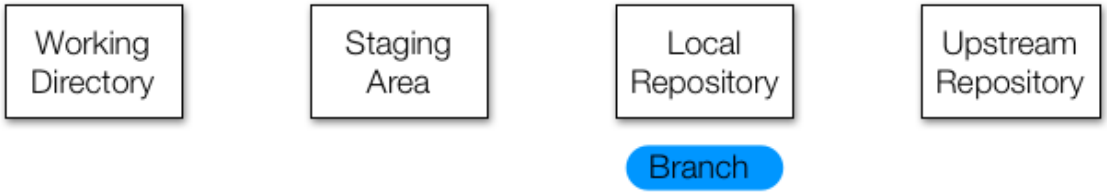
[4]:
```
%%bash
cd /tmp/cs207_david_sondak
git branch -avv
```

```
* master                    f617a3f [origin/master] FALL 2018 materials.
  remotes/origin/HEAD       -> origin/master
  remotes/origin/ad-project 7574906 Printing out lots of decimal places in the
root.
  remotes/origin/gh-pages   36aa8a1 Shifted schedule around.
  remotes/origin/master     f617a3f FALL 2018 materials.
  remotes/origin/tim        4bc23fb Added MathCS207.py
```

All of these branches are nothing but commit-streams in disguise, as can be seen above. It's a very simple model that leads to a lot of interesting version control patterns.

Since branches are so light-weight, the recommended way of working on software using `git` is to create a new branch for each new feature you add, test it out, and if good, merge it into master. Then you deploy the software from `master`. We have been using branches under the hood. Let's now lift the hood.

1.5.1 `branch`



Branches can also be created manually, and they are a useful way of organizing unfinished changes.

The `branch` command has two forms. The first:

```
git branch
```

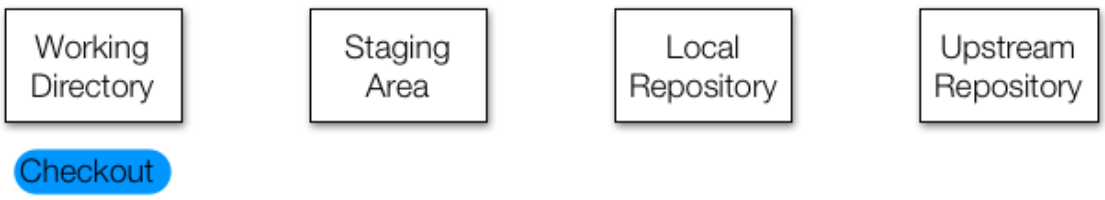
simply lists all of the branches in your local repository. If you run it without having created any branches, it will list only one, called `master`. This is the default branch. You have also seen the use of `git branch -avv` to show all branches (even remote ones).

The other form of the `branch` command creates a branch with a given name:

```
git branch branch_name
```

It's important to note that this new branch is not *active*. If you make changes, those changes will still apply to the `master` branch, not `branch_name`. That is, after executing the `git branch branch_name` command you're still on the `master` branch and **not** the `branch_name` branch. To change this, you need the next command.

1.5.2 `checkout`



`Checkout` switches the active branch. Since branches can have different changes, `checkout` may make the working directory look very different. For instance, if you have added new files to one

branch and then check another branch out, those files will no longer show up in the directory. They are still stored in the `.git` folder, but since they only exist in the other branch, they cannot be accessed until you check out the original branch.

```
git checkout my-new-branch
```

You can combine creating a new branch and checking it out with the shortcut:

```
git checkout -b my-new-branch
```

Try this out on your course repository.

[5]:
```
%%bash
cd /tmp/cs207_david_sondak
git branch lecture4_exercise
```

See what branches we have created.

[6]:
```
%%bash
cd /tmp/cs207_david_sondak
git branch
```

```
  lecture4_exercise
* master
```

Notice that you have created the `lecture4_exercise` branch but you're still *on* the `master` branch.

Jump onto the `lecture4_exercise` branch.

[7]:
```
%%bash
cd /tmp/cs207_david_sondak
git checkout lecture4_exercise
git branch
```

```
* lecture4_exercise
  master
Switched to branch 'lecture4_exercise'
```

Notice that it is bootstrapped off the `master` branch and has the same files. You can check that with the `ls` command.

[1]:
```
%%bash
cd /tmp/cs207_david_sondak
ls
```

```
LICENSE
README.md
homework
lectures
legacy
notes
```

```
project
supplementary-python
```

Note: You could have created this branch and switched to it all in one go by using `git checkout -b lecture4_exercise`.

Now let's check the status of our repo.

[9]:
```
%%bash
cd /tmp/cs207_david_sondak
git status
```

```
On branch lecture4_exercise
nothing to commit, working tree clean
```

Alright, so we're on our new branch but we haven't added or modified anything yet; there's nothing to commit.

1.5.3 Adding a file on a new branch

Let's add a new file. Note that this file gets added on this branch only! Notice that I'm still using the `echo` command. Once again, this is only because `jupyter` can't work with text editors. If I were you, I'd use `vim`, but you can use whatever text editor you like.

[10]:
```
%%bash
cd /tmp/cs207_david_sondak
echo '# Things I wish G.R.R. Martin would say:  Finally updating A Song of Ice␣
↪and Fire.' > books.md
git status
```

```
On branch lecture4_exercise
Untracked files:
  (use "git add <file>…" to include in what will be committed)

        books.md

nothing added to commit but untracked files present (use "git add" to track)
```

We add the file to the index, and then commit the files to the local repository on the `lecture4_exercise` branch.

[11]:
```
%%bash
cd /tmp/cs207_david_sondak
git add books.md
git status
```

```
On branch lecture4_exercise
Changes to be committed:
  (use "git reset HEAD <file>…" to unstage)
```

```
        new file:   books.md
```

[12]:
```
%%bash
cd /tmp/cs207_david_sondak
git commit -am "Added another test file to demonstrate git features"
git status
```

```
[lecture4_exercise 06c0fbd] Added another test file to demonstrate git features
 1 file changed, 1 insertion(+)
 create mode 100644 books.md
On branch lecture4_exercise
nothing to commit, working tree clean
```

Pause: Make sure you really understand what the `-am` option does!

At this point, we have committed a new file (`books.md`) to our new branch in our local repo. Our remote repo is still not aware of this new file (or branch). In fact, our `master` branch is still not really aware of this file.

Note: There are really two options at this point: 1. Push the current branch to our upstream repo. This would correspond to a "long-lived" branch. You may want to do this if you have a version of your code that you are maintaining. 2. Merge the new branch into the local master branch. Depending on your chosen workflow, this may happen *much* more frequently than the first option. You'll be creating branches all the time for little bug fixes and features. You don't necessary want such branches to be "long-lived". Once your feature is ready, you'll merge the feature branch into the `master` branch, `stage`, `commit`, and `push` (all on `master`). Then you'll delete the "short-lived" feature branch.

We'll continue with the first option for now and discuss the other option later.

1.5.4 Long-lived branches

Ok, we have committed. Lets try to push!

[14]:
```
%%bash
cd /tmp/cs207_david_sondak
git push
```

```
fatal: The current branch lecture4_exercise has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin lecture4_exercise



␣
↪---------------------------------------------------------------------------

       CalledProcessError                        Traceback (most recent call␣
↪last)
```

```
      <ipython-input-14-5c7726728118> in <module>
   ----> 1 get_ipython().run_cell_magic('bash', '', 'cd /tmp/
↪cs207_david_sondak\ngit push\n')


      ~/anaconda3/lib/python3.6/site-packages/IPython/core/interactiveshell.py␣
↪in run_cell_magic(self, magic_name, line, cell)
      2319             magic_arg_s = self.var_expand(line, stack_depth)
      2320             with self.builtin_trap:
   -> 2321                 result = fn(magic_arg_s, cell)
      2322             return result
      2323


      ~/anaconda3/lib/python3.6/site-packages/IPython/core/magics/script.py in␣
↪named_script_magic(line, cell)
      140             else:
      141                 line = script
   --> 142             return self.shebang(line, cell)
      143
      144         # write a basic docstring:


      <decorator-gen-109> in shebang(self, line, cell)


      ~/anaconda3/lib/python3.6/site-packages/IPython/core/magic.py in␣
↪<lambda>(f, *a, **k)
      185     # but it's overkill for just that one bit of state.
      186     def magic_deco(arg):
   --> 187         call = lambda f, *a, **k: f(*a, **k)
      188
      189         if callable(arg):


      ~/anaconda3/lib/python3.6/site-packages/IPython/core/magics/script.py in␣
↪shebang(self, line, cell)
      243             sys.stderr.flush()
      244         if args.raise_error and p.returncode!=0:
   --> 245             raise CalledProcessError(p.returncode, cell, output=out,␣
↪stderr=err)
      246
      247     def _run_script(self, p, cell, to_close):
```

```
        CalledProcessError: Command 'b'cd /tmp/cs207_david_sondak\ngit push\n''␣
↪returned non-zero exit status 128.
```

Fail! Why? Because `git` didn't know what to push to on `origin` (the name of our remote repo) and didn't want to assume we wanted to call the branch `lecture4_exercise` on the remote. We need to tell that to `git` explicitly (just like it tells us to).

[15]:
```
%%bash
cd /tmp/cs207_david_sondak
git push --set-upstream origin lecture4_exercise
```

```
Branch 'lecture4_exercise' set up to track remote branch 'lecture4_exercise'
from 'origin'.

remote:
remote: Create a pull request for 'lecture4_exercise' on GitHub by visiting:
remote:
https://github.com/dsondak/cs207_david_sondak/pull/new/lecture4_exercise
remote:
To https://github.com/dsondak/cs207_david_sondak.git
 * [new branch]      lecture4_exercise -> lecture4_exercise
```

Aha, now we have both a remote and a local for `lecture4_exercise`. We can use the convenient arguments to `branch` in order to see the details of all the branches.

[16]:
```
%%bash
cd /tmp/cs207_david_sondak
git branch -avv
```

```
* lecture4_exercise                06c0fbd [origin/lecture4_exercise] Added
another test file to demonstrate git features
  master                           f617a3f [origin/master] FALL 2018 materials.
  remotes/origin/HEAD              -> origin/master
  remotes/origin/ad-project        7574906 Printing out lots of decimal places
in the root.
  remotes/origin/gh-pages          36aa8a1 Shifted schedule around.
  remotes/origin/lecture4_exercise 06c0fbd Added another test file to
demonstrate git features
  remotes/origin/master            f617a3f FALL 2018 materials.
  remotes/origin/tim               4bc23fb Added MathCS207.py
```

We make sure we are back on master

[17]:
```
%%bash
cd /tmp/cs207_david_sondak
git checkout master
```

```
Your branch is up to date with 'origin/master'.

Switched to branch 'master'
```

What have we done?

We created a new local branch, created a file on it, created that same branch on our remote repo, and pushed all the changes. Finally, we went back to our `master` branch to continue work there.

1.5.5 Deliverables

The deliverables for this part are the `lecture4_exercise` branch in the remote course repo.

Warning: You shouldn't `push` to `master` for this excercise because the file that you created isn't really needed in your course repo.

1.5.6 Short-lived branches

Now we'll look into option 2 above. Suppose we want to add a feature to our repo. We'll create a new branch to work on that feature, but we don't want this branch to be long-lived. Here's how we can accomplish that.

We'll go a little faster this time since you've seen all these commands before. Even though we're going a little faster this time, make sure you understand what you're doing! Don't just copy and paste!!

[18]:
```
%%bash
cd /tmp/cs207_david_sondak
git checkout -b feature-branch
```

```
Switched to a new branch 'feature-branch'
```

[19]:
```
%%bash
cd /tmp/cs207_david_sondak
git branch
```

```
* feature-branch
  lecture4_exercise
  master
```

1.5.7 WARNING

Do not copy the next several lines! The example I'm giving here will be slightly different from what you will do (although all the steps will be the same).

Your requirements Edit your `README.md` by including a subsection (level 2 Markdown header) called `Content`. Under the `Content` section, create a bulleted list where each bullet is a directory name (e.g. `lectures`, `homework`, etc). Provide a one-sentence description next to each bullet title.

The difference between what I do below and what you will do is that you will be working with the `README.md` file and I created a new file called `feature.txt`.

[20]:
```
%%bash
cd /tmp/cs207_david_sondak
echo '# The collected works of G.R.R. Martin.' > feature.txt
```

[21]:
```
%%bash
cd /tmp/cs207_david_sondak
git status
```

```
On branch feature-branch
Untracked files:
  (use "git add <file>…" to include in what will be committed)

        feature.txt

nothing added to commit but untracked files present (use "git add" to track)
```

[22]:
```
%%bash
cd /tmp/cs207_david_sondak
git add feature.txt
git commit -m 'George finished his books!'
```

```
[feature-branch a10e6ec] George finished his books!
 1 file changed, 1 insertion(+)
 create mode 100644 feature.txt
```

At this point, we've committed our new feature to our feature branch in our local repo. Presumably it's all tested and everything is working nicely. We'd like to merge it into our `master` branch now. First, we'll switch to the `master` branch.

[2]:
```
%%bash
cd /tmp/cs207_david_sondak
git checkout master
ls
```

```
Your branch is up to date with 'origin/master'.
LICENSE
README.md
homework
lectures
legacy
notes
project
supplementary-python
Switched to branch 'master'
```

The `master` branch doesn't have any idea about our new feature yet! We should merge the feature branch into the `master` branch.

[24]:
```
%%bash
cd /tmp/cs207_david_sondak
git merge feature-branch
```

```
Updating f617a3f..a10e6ec
```

```
Fast-forward
 feature.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 feature.txt
```

[3]:
```
%%bash
cd /tmp/cs207_david_sondak
git status
ls
```

```
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
LICENSE
README.md
feature.txt
homework
lectures
legacy
notes
project
supplementary-python
```

Now our `master` branch is up to date with our feature branch. We can now delete our feature branch since it is no longer relevant.

[26]:
```
%%bash
cd /tmp/cs207_david_sondak
git branch -d feature-branch
```

```
Deleted branch feature-branch (was a10e6ec).
```

Finally, let's push the changes to our remote repo.

[ ]:
```
%%bash
cd /tmp/cs207_david_sondak
git push
```

Great, so now you have a basic understanding of how to work with branches. There is much more to learn, but these commands should get you going. You should really familiarize yourself with Chapter 3 of the Git book for more details and workflow ideas.

1.5.8 Deliverables

For this part, your deliverable is an updated `README.md` file on your `master` branch in your remote course repo.

1.6 Merge Conflicts

You already have experience with merge conflicts from the first homework assignment. Let's do it one more time under supervision, just a bit faster this time. After this, you're on your own.

1.6.1 Setting up

Choose a partner to work with. Please add them as a collaborator on your `playground` repo. One of you should clone the other's playground repo. For example, suppose Sally and Joe decide to work together. Together, they decide that Joe will clone Sally's playground repo. This should be done in your `Classes/CS207/` directory somewhere. It's up to you where, just try to be organized.

1.6.2 Making Some Changes

Each partner should make some changes to the playground repo. The easiest way to do this to start will be to change the title of the `README.md` file. For the sake of the exercise, each partner should make the title of the `README.md` file something different. Feel free to do something more creative / interesting.

Reminder: One partner will be working from their original playground repo while the other partner will be working from the newly cloned playground repo.

Once you've each made the changes, please try to stage-commit-push. This should work without a problem for one of the partners. The other partner should get an error similar to:

```
To https://github.com/dsondak/playground.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/dsondak/playground.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

The second partner realizes that they've made a mistake. **Always** fetch and merge (or pull) from the remote repo before doing your work for the day or pushing your recent changes. However, perhaps you're a little nervous since it only took you a minute to make your changes. You should realize that someone else probably did a push in the meantime. Nevertheless, you decide to proceed.

1.6.3 Resolving Some Conflicts

The partner who failed to push should try to do a `fetch-merge` sequence. Try out the following command:

```
git fetch
git merge origin/master
```

Think about what these commands are doing! Can you explain them?

You should get an error message similar to:

```
Auto-merging intro.md
CONFLICT (add/add): Merge conflict in intro.md
```

```
Automatic merge failed; fix conflicts and then commit the result.
From https://github.com/dsondak/playground
   999fd74..2658cab  master     -> origin/master
```

There is a conflict in `intro.md` [in this particular example, but yours might be slightly different] and `git` can't figure out how to resolve the conflict automatically. It doesn't know who's right. Instead, `git` produces a file that contains information about the conflict.

Do a `cat` on the conflicted file. You should see something like:

```
<<<<<<< HEAD
# A Project by Joe
=======
# A Project by Sally
>>>>>>> origin/master
```

The partner with the conflict knows that their partner is working on the same project as they are (they're teammates) so don't be alarmed.

Resolve the conflict Talk to your partner and decide on a new title for the `README.md` file. Update the file and do a stage-commit-push sequence (with a good commit message!!) to update the remote repo.

The merge conflict has been resolved! Of course, the other parter's local repo doesn't yet know about what just happened. They need to `fetch` and `merge` to get the updates. **Do this now!**

You should see output similar to:

```
Updating 2658cab..51c6b05
Fast-forward
 intro.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
From https://github.com/dsondak/playground
   2658cab..51c6b05  master     -> origin/master
```

Observe: This is reporting a *fast-forward* operation. Why wasn't there another merge conflict? Please explain.

1.6.4 Comments

So what actually happened here? [Note: We'll use Joe and Sally to explain what's going on for clarity.]

And as expected, `git` knows how to resolve this conflict. The reason `git` can resolve this conflict even though the files differ on the same line is that `git` has the commit history, too. When Sally made their original commit, they were given a commit hash (starting with 2658cab). When Joe resolved the merge conflict, Joe created a new commit hash (51c6b05) which unified the changes in commit 2658cab (Sally's original commit) and commit 3b934ee (Joe's original commit). Then, when Joe pushed, all of this information was given to the upstream repository. So `git` has a record stating that the merge resolution commit 51c6b05 is a subsequent commit to Sally's original changes in 2658cab. When Sally `fetch`ed the upstream repo, Sally got this information, too. So

when Sally executed a `merge`, Sally was merging a predecessor (2658cab) with its direct successor (51c6b05), which `git` handles simply by using the successor.

The tricky conflict resolution that Joe did was effectively a way of taking two separate branches and tying them together.

One more note on binary files A problem that students sometimes run into occurs when they try to update their local repo from another repo *after* some changes to a `.pdf` file have been made.

One of the big lessons here is that versioning binary files with `git` requires some special tools. In this case, the binary file was a `.pdf` document. In another case it may be an executable file.

The reason why binary files are difficult to version is because `git` must store the entire file again after each commit. This is essentially a consequence of the fact that there is no clear way to `diff` binary files. Hence, the `merg`ing operation has problems.

There is extensive information around for the special tools `git` has for working with binary files. For the particular case of `.pdf` files, you can use some special arguments to the `git checkout` command. A nice discussion can be found at https://stackoverflow.com/questions/278081/resolving-a-git-conflict-with-binary-files.

My recommendation is that you try to stay away from versioning binary files. I put them up on `git` because I will not be changing the lectures slides much (if at all) over the course of the semester and because the lecture slides do not take up much space (and will therefore not have much of an effect on the speed of `git`).

1.7 Git habits

** * Commit early, commit often. * **

Git is more effective when used at a fine granularity. For starters, you can't undo what you haven't committed, so committing lots of small changes makes it easier to find the right rollback point. Also, merging becomes a lot easier when you only have to deal with a handful of conflicts.

** * Commit unrelated changes separately. * **

Identifying the source of a bug or understanding the reason why a particular piece of code exists is much easier when commits focus on related changes. Some of this has to do with simplifying commit messages and making it easier to look through logs, but it has other related benefits: commits are smaller and simpler, and merge conflicts are confined to only the commits which actually have conflicting code.

** * Do not commit binaries and other temporary files. * **

Git is meant for tracking changes. In nearly all cases, the only meaningful difference between the contents of two binaries is that they are different. If you change source files, compile, and commit the resulting binary, git sees an entirely different file. The end result is that the git repository (which contains a complete history, remember) begins to become bloated with the history of many dissimilar binaries. Worse, there's often little advantage to keeping those files in the history. An argument can be made for periodically snapshotting working binaries, but things like object files, compiled python files, and editor auto-saves are basically wasted space.

** * Ignore files which should not be committed * **

Git comes with a built-in mechanism for ignoring certain types of files. Placing filenames or wildcards in a `.gitignore` file placed in the top-level directory (where the `.git` directory is also located) will cause git to ignore those files when checking file status. This is a good way to ensure you don't commit the wrong files accidentally, and it also makes the output of `git status` somewhat cleaner.

** * Always make a branch for new changes * **

While it's tempting to work on new code directly in the `master` branch, it's usually a good idea to create a new one instead, especially for team-based projects. The major advantage to this practice is that it keeps logically disparate change sets separate. This means that if two people are working on improvements in two different branches, when they merge, the actual workflow is reflected in the git history. Plus, explicitly creating branches adds some semantic meaning to your branch structure. Moreover, there is very little difference in how you use git.

** * Write good commit messages * **

I cannot understate the importance of this.

** Seriously. Write good commit messages. **