Git(hub) Training

Installing git

Windows: https://git-scm.com/download/win

Mac: Git will be automatically installed when you first use it

Linux: `sudo apt install git`

Basic ideas

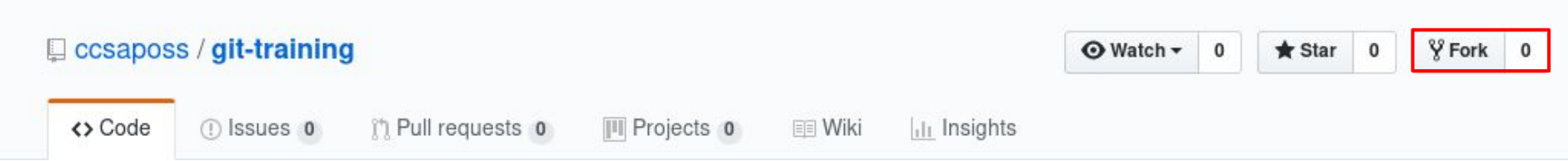
- Repository
  - A bunch of code that accomplishes some project or goal (i.e. an app)
- Commit
  - A set of **changes** made to the code
- Branch
  - A version of the repository where some feature is being implemented/tested
- Fork
  - Someone’s copy of a repository on their own Github account
- Remote
  - Link to fork of a repository, allows access to multiple forks

Basic ideas (cont.)

- HEAD
  - Your ‘location’ in a branch, usually the most recent commit
- Index
  - Changes that are ready to be committed (staged)
- Working directory/tree
  - Where you are now, including changes that haven’t yet been ‘staged’

Make a fork

- Fork this repository for the training today
  - Go to `github.com/ccsaposs/git-training`
  - Click the “Fork” button in the top right



Setting up

- Clone your forked repo (copy from github to your computer)

```
> git clone https://github.com/[your-username]/git-training
```

- Navigate into the local repo

```
> cd git-training
```

- Add `upstream` remote (the main codebase)

```
> git remote add upstream https://github.com/ccsaposs/git-training
```

- Fetch from the `upstream` remote

```
> git fetch upstream
```

    - Updates information from the main repo so you know what branches you have

Remotes

- Link between git and Github
  - Lets you get (fetch) code from github and send (push) code to github
- Naming conventions
  - `upstream` - the repo you forked
    - Usually `frc1678`
    - In this case, `ccsaposs`
  - `origin` - your fork
  - `<first-name>` - buddy forks (other people in your app group)
    - e.g. “`carl`”
- List all remotes: `git remote`

First change

- Open `SPR.py`
  - Change the range for “randint” from (1,5) to (1,8)
  - Save
- Check `git status`, check `git diff`

```
> git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

      modified:   SPR.py

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

Second change

- Open `server.py`
  - Change the time.sleep() from 5 to 15
  - Save
- Check `git status`, check `git diff`

```
> git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

      modified:   SPR.py
      modified:   server.py

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

First commit

- Stage `SPR.py` and `server.py` for commit
  - When we commit, only the staged changes will be included
  - `git add SPR.py`
  - `git add server.py`
- Check `git status`

```
> git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

      modified:   SPR.py
      modified:   server.py
```

First commit

- Wait! We don’t want `SPR.py` and `server.py` in the same commit!
  - They’re separate changes
  - We want to keep our commits as small as possible
  - Should the following be separated?
    i. Moving code around, updating comments
    ii. Renaming a file, deleting commented out code
    iii. Changing an import in two different files
- We want to unstage `server.py`
  - `git reset HEAD server.py`

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

First commit

- `git status`

```
> git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

      modified:   SPR.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

      modified:   server.py
```

First commit

- Create a commit
  - `git commit`
  - Type a good commit message
    - Subject line - 50 characters or less
    - Extended description should be included unless the change is very simple
      - Each line should be 72 characters or less

Example commit message

```
Modify 'Aakash' SPR multiplier.

Change random integer possible range from (1,5) to (1,8) for 'Aakash'
SPR multiplier.
```

50

(separate w/blank line)

72

Second commit

- Let's add the `server.py` changes
  - `git add server.py`
  - `git commit`
- What is a good commit message?
- Check `git log`
  - Shows history of commits

Third commit

- Open `SPR.py`
  - We don’t want to display SPRs
  - In the print statement, replace the SPR with “redacted”.
  - Add and commit changes
- What is a good commit message?

Notes on commits

- Use the imperative mood (i.e. “Change this thing”, “Add this feature”)
- Be extremely specific about changes you made, if they don’t fit into a subject line, use an extended description
- Amending the most recent commit message

```
> git commit --amend
```

- Should I amend commit messages already on Github?
  - **No.** This changes the entire history of the repository. That means that anyone else working on this code will have to manually change their history, which should not be their responsibility
- Make sure your commits are small, so only add a few changes at a time. This leaves a more detailed description for anyone else working on the branch
- Changing commit text editor

```
> git config --global core.editor “[text editor]”
```

Branches

- Check `git status`

```
> git status
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

What does “On branch master” mean?

Branches

- A branch is a version of the repository
  - Branches are used to work on multiple, unrelated changes at the same time
    - e.g. Updating SPR calculations and changing timing in server.py
- Branches can have different commits
- Branches should only have 1 feature
  - Separate unrelated changes into different branches
  - Why? Easier to collaborate, review, and fix problems

Branches

- The `master` branch should **never** be committed to.
- Oh no! We've already made 3 commits on the `master` branch.
- How should we have done it?
  - 2 branches, one for server.py, one for SPR.py

- Let's look at how we should have done it:
  - Before we create a branch, we want to make sure we have the latest code

Updating code

- Fetching

```
> git fetch upstream
```

  - Updates information from a remote
- Merging

```
> git merge upstream/master
```

  - Combines the master branch of the upstream remote with your HEAD
- Pulling

```
> git pull upstream master
```

  - Shorthand for the above two commands, will merge remote code with local code

Branches

- Create branch

```
> git branch [branch-name]
```

- Switch to branch

```
> git checkout [branch-name]
```

- List branches

```
> git branch
```

- Delete branch

```
> git branch -d [branch-name]
```

Tip:

```
> git checkout -b [branch-name]
```

will create a new branch and switch to it

Branches

- How do we fix this?
  - What do we need to do?
    - Create a “`SPR.py`” branch
      - Move the 1st and 3rd commits we made to this branch
    - Create a “`server.py`” branch
      - Move the 2nd commit we made to this branch
    - Restore the “`master`” branch to match upstream

Branches

- Let’s start with by moving the 2nd commit to the “`server.py`” branch
- We need to start our branch based on upstream/master
  - `git checkout upstream/master`
  - Create and checkout a new branch called “`server.py`”
  - To move the commit, we can “cherry-pick” it

Cherry picking!

- Super useful command

```
> git cherry-pick [commit]
```

    - Takes a single commit and moves it to the current branch
        - Specific a commit using its hash (which can be found using `git log`)

Don’t be afraid to cherry-pick!

- Let’s cherry-pick that 2nd commit
    - Check `git log` afterwards to see if you did it correctly

Branches

- Now we need to isolate the 1st and 3rd commits in the “`SPR.py`” branch
- Let’s start with all 3 of our commits
  - `git checkout master`
  - Create and checkout a new branch called “`SPR.py`”
  - We can ‘re-arrange’ the commits using `git rebase [commit] -i`
    - Change the 2nd commit from “`pick`” to “`drop`”
    - Save and close
  - Check `git log`

Branches

- Last step: reset the `master` branch to match `upstream/master`
- First, switch to the `master` branch
- We can use `git reset`

Resetting

- Soft resetting

```
> git reset --soft
```

  - Un-commits previous commit, files are still changed and staged for change, but changes are no longer part of the commit history

- Mixed resetting

```
> git reset --mixed
```

  - Goes one step further, unstages commits, but changes are still there in working directory

- Hard resetting ***HERE BE DRAGONS***

```
> git reset --hard
```

  - **Destroys** changes, they no longer exist, completely resets to previous commit

Branches

- Last step: reset the `master` branch to match `upstream/master`
- First, switch to the `master` branch
- We can use `git reset`
  - Which one do we want to use? (hard, soft, or mixed)
  - `git reset --hard upstream/master`

- Next, let's upload our changes to github
  - So far, all of this has been local to our computer

Changing code on github

- Pushing code

```
> git push [remote name] [branch-name]
```

- Making a PR!
  - On github, navigate to your fork. If you recently pushed, there should be a popup asking if you want to make a new PR
  - If you didn't recently push or you're creating a PR from someone else's fork:
    - Head to the main repository and click "New Pull Request". Then specify the fork and branch you want to use.
  - Good code review will be covered outside of this training
- Adding to a PR
  - Just make new commits and push to the same branch on the same remote, your PR will update automatically

More notes on pushing

- Pushing from a specific branch

```
> git push [branch-name]:[branch-name]
```

  - First branch is what you want to push from, second is what you want to push to
    - Use this if your branch name doesn't match the branch name on github
- Deleting a branch

```
> git push :[branch-name]
```

  - Same as above, but you're pushing nothing to the branch, wiping it clean

Code review

- Let’s practice our new code review process
  - On github, we have labels to mark if a PR (pull request) needs review
    - On your PR, add the `needs review` label
- Look at the scout-QR-2019 `README.md` for a recap of the process
- First, pair up with someone for the buddy review
  - Adding your review to github
    - Click “Files Changed”
    - Click “Review changes”
- After everyone’s done, you’ll pair up with a different person for the peer review
  - Add your review to github

Other remotes

- We want to get code from David's fork
- First, add his repo as a remote
  - `git remote add david https://github.com/daviji/git-training`
  - `git fetch david`
- Let's checkout his `function-rename` branch
  - `git checkout function-rename`
- David's last commit broke the server
  - How do we fix it?
    - We can't remove the commit, it's already on github
    - Let's revert it

Reverting

- Revert to commit/branch

```
> git revert [commit ref]
```

  - “Undo” for the commit, creates a new revert commit
  - Allows you to go back to previous commits
- Using checkout

```
> git checkout [commit ref] -- [file name]
```

  - Doesn’t create revert commit
  - Simply takes you back to previous commit, future changes still saved
  - File name is optional, will revert only that file

Let’s make one last change

- In SPR.py, replace “`random.randint()`” with “8”
- Check `git status`

```
> git status
On branch function-rename
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

     modified:   SPR.py

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

Let’s make one last change

- Oh no! We’re on the wrong branch!
- How do we take our changes over to a different branch?
  - We haven’t committed yet
  - We can do git stash to save uncommitted changes
-
```
> git stash
```
  - “Stores” changes without committing, you can then navigate and apply changes to wherever. Be careful with this, as it is working with unsaved changes

Finishing up

- Create a new branch to put the stashed changes on
  - What do we need to do first?
- Once you have a branch, use `git stash pop` to “release” the changes
- Push this to Github and create a PR
  - We won’t review for this one

Professionalism with git(hub)

- All commit messages should be strictly professional
  - Both in the subject line and extended description
  - We want the information to be concise and as useful as possible
  - Includes PR commits
- Code review/testing messages
  - When approving, not as strict
    - Include the necessary information **first**
      - (e.g. “Buddy review completed” or “User testing successful”)
    - You can (but do not need to) add other comments after
      - (e.g. “Nice job!” or “Good to finally get this done!”)
  - When requesting changes, keep the information concise + useful
- Keep in mind that 1678 is the go-to example in FRC for electronic scouting
  - We open-source our code, lots of teams will be able to see these messages

Any Questions?

Feel free to ask me or Carl if anything else comes up