gwu-libraries.github.io

Git best practices and tips

General practices ——————————-

Create (and check out) a local branch to track a remote branch (i.e. a branch on the github server)

```
git checkout -b mybranch origin/myremotebranch
```

Best practice would be to name mybranch exactly the same as myremotebranch, unless there is a good reason to do otherwise.

Push a local branch to remote (and track)

```
git push -u origin mybranch
```

Rebasing periodically while working on a branch

This is a preferred practice for bringing the latest changes from master instead of using merge because it leaves a cleaner history.

Be very careful about rebasing public branches. Since it gets rid of some commits, it can cause problems for other developers doing work on the branch.

```
# in your local branch MYBRANCH
git add STUFF
git commit -m 'NOTE'
# repeat as desired

# IN THE MEANWHILE… the remote master branch (origin/master) has had
# some changes.

# grab the latest stuff from origin/master to update your
# local master branch
git checkout master
```

```
git pull origin master

# go back to MYBRANCH and now rebase with the changes in your
# local master branch
git checkout MYBRANCH
git rebase master

# if you encounter merge conflicts… edit each affected file, then
# git add that file.  Then:  git rebase --continue

# commit your updated branch to origin.  May require a -ff to force.
git push origin MYBRANCH
```

Git squashing before committing a branch to master

The effect of squashing is so that when the branch is merged, it's one commit rather than a series of little commits. We want to keep the commit log of master clean and readable.

`git commit --squash` gives you a chance to compose a single commit message, so you'll want to think ahead about what that message should be. It should describe what changes this commit entails.

Here is a recommended series of steps to follow:

```
# review the last several commits on this branch

git log

# let's say you did that and you determined that you want to
# squash together the last 12 commits.
# Reset the current branch to the commit just BEFORE the last 12 (or other number)

git reset --hard HEAD~12

# Alternatively, you could have identified the commit hash of the last commit BEFOI
# changes you want to squash.  That would look like:
# git reset --hard a7b5c7302c22fb967bba50e55a5fcc7bd2d26cf0

# Next:  HEAD@{1} is where the branch was just before the previous command.
# This next command sets the state of the index to be as it would just
# after a merge from that commit:

git merge --squash HEAD@{1}

# Commit those squashed changes.  The commit message will be
```

```
# prepopulated with a concatenation of the commit messages of all
# the squashed commits.
# You will be placed in the editor, where you can compose one
# unified commit annotation.
# Recommend that you delete the prepopulated string of commit
# messages, and compose one clean message describing the changes
# in the branch.

git commit

# push the branch to the remote repo.
# Seems to require -ff to force it.
git push -ff origin MYBRANCH
```

Helpful practices

See previous commit

```
git show HEAD^
```

Compare commits across branches

```
# Commits in experiment, but not master
git log master..experiment

# Opposite
git log experiment..master
```

Interactive staging

```
git add -i
```

Saving dirty state and return to clean

Save the dirty state of your working directory and return to a clean state. State can be re-applied to same or a different branch.

```
git stash save mychanges
git checkout anotherbranch
```

```
git stash pop
```

Useful flags for `git stash`:

- `--keep-index`: Don't stash staged files.
- `-u`: Stash untracked files as well.

Remove untracked files and directories

```
git clean -f -d
```

Useful flags for `git clean`:

- `-x`: Also clean ignored files.
- `-i`: Interactive clean.

Search for code

```
git grep -n somfunct
```

Search for commits with code

```
git log -Ssomefunct --oneline
```

History of changes for code

```
git log -L :somefunct:somefile.py
```
