Review Questions

This document contains review questions for Git basics.

Instructions

Overview

* **Problem**: Consider the example from the lecture, where we created a branch for our data analysis. Why did we create a new branch for this? Why _not_ simply do this on `master`?

* **Solution**: Doing the work directly on `master` would make it harder to keep that work organized. Branches let us keep our work sandboxed and organized.

* **Problem**: Write two advantages to creating branches instead of working directly on `master`.

* **Solution**:

  1. We can isolate our experiments to a single branch—if we break something on our `data_analysis` branch, we at least know that the code on `master` still works.

  2. We can focus on writing _one_ new feature at a time, instead of having work for a handful of different features and bugfixes in a single branch.

Branching

For the problems below, consider the following commit history:

  ```bash
  (master) | [m1] -> [m2] -> [m3] -> [m4]
  ```

* **Problem**: Draw a new branch, called `plotting_data`. It should branch from the second commit to `master`.

* **Solution**:

  ```bash
  (master)        | [m1]-[m2]
                           \
  (plotting_data) |         \-[pd1]-[pd2]
  ```

* **Problem**: When you first create `plotting_data`, are the files on that branch the same as the files in `[m1]`? In `[m2]`? Why, or why not?

* **Solution**

The files on `plotting_data` are the same as the files in `[m2]`.

This is because we created and checked out `plotting_data` while we were on `[m2]`. This means our files will look like they did when we last committed to `[m2]`, but that Git tracks any further changes to files on `plotting_datas`, _not_ `master`.

* **Problem**: Add two commits to the `master` branch.

* **Solution**:

  ```bash
  (master)        | [m1]-[m2]-[m3]-[m4]
                           \
  (plotting_data) |         \-[pd1]-[pd2]
  ```

* **Problem** Add two commits to the `plotting_data` branch, named `[pd1]` and `[pd2]`.

* **Solution**

  ```bash
  (master)        | [m1]-[m2]-[m3]-[m4]
                           \
  (plotting_data) |         \-[pd1]-[pd2]
  ```

* **Problem**: Are the files in `[pd1]` and `[m3]` the same? Why, or why not?

* **Solution**: No. The code in `[pd1]` and `[m3]` are _not_ the same. Any two given branches should contain _different_ work, so `[m3]` probably contains a patch or bugfix totally unrelated to `[pd1]`.

Merging

* **Problem**: Merge `[pd2]` with `master`.

* **Solution**:

  ```bash
  (master)        | [m1]-[m2]-[m3]-[m4]- - --[m5]
                           \                /
  (plotting_data) |         \-[pd1]-[pd2]-/
  ```

* **Problem**: Explain how this merge changes the files in `master`.

* **Solution**: `master` now has the most recent changes to `clean_data.py` made on _either_ the `master` or `plotting_data` branch. It will also have the new files add in the `plotting_data` branch.

* **For the problems below, consider the following history.**

  ```bash
  (master)        | [m1]-[m2]-[m3]-[m4]- - -- - -- - -[m5]
                             \               / (M)
  (plotting_data) |           \-[pd1]-[pd2]-/
  ```

  * Assume `[m4]` on `master` updates `clean_data.py`, but doesn't change the directory structure.

  * Assume the root project directory looks as follows at each commit:

    ```bash
    [m4]
    root/
      |_analyze_data.py
      |_clean_data.py
      |_output/
        |_cleanedRideData.csv
      |_Resources/
        |_rideData.csv

    [pd2]
    root/
      |_analyze_data.py
      |_clean_data.py
      |_helpers.py
      |_plot_data.py
      |_output/
        |_cleanedRideData.csv
        |_plots.pdf
```

```
[pd2]
root/
  |_analyze_data.py
  |_clean_data.py
  |_helpers.py
  |_plot_data.py
  |_output/
    |_cleanedRideData.csv
    |_plots.pdf
  |_Resources/
    |_rideData.csv
```

* **Problem**:

  1. When we merge `master` and `plotting_data`, which version of each file do we get?

  2. Draw the directory structure for the last commit to `master`—after the merge—and label each file with the branch it comes from. Assume that the only files changed on `plotting_data` were `helpers.py` and `plot_data.py`.

* **Solution**:

  ```bash
  [m5]
  root/
    |_analyze_data.py (master)
    |_clean_data.py (master)
    |_helpers.py (plot_data)
    |_plot_data.py (plot_data)
    |_output/ (mixed)
      |_cleanedRideData.csv (master)
      |_plots.pdf (plot_data)
    |_Resources/ (master)
      |_rideData.csv
  ```

- - -

Branch Demo

0. Getting the Repo

Before we can work with Git, we must either **create a new repository**, or **clone one from GitHub**.

Note that, in the examples below, we use `git status` before every `git commit`. This is a best practice that helps ensure a deliberate commit history. For brevity's sake, this line will be omitted in future files, **but assume we've always run `git status` before any `git commit`**.

Clone from GitHub

If someone has already shared a repository on GitHub, you can **clone** it to your local machine with \`git.

```bash
# Clone an existing repo.
git clone <repo_url>
# Navigate into newly created repo directory
cd <repo_name>
```

1. Add Files

Next, we simply develop as normal, and `commit` our changes whenever we make significant progress.

In general, it's best to **commit early** and **commit often**. Frequent snapshots ensure you'll never be far away from a "last working version".

```bash
# Create a file, called clean_data.py
touch clean_data.py

# Add and commit clean_data.py...
git add clean_data.py
git status
git commit -m "First commit."

# Add cleanup code to clean_data.py...
git add clean_data.py
git status
git commit -m "Clean up provided data."

# Add code to export clean data...Note that `add .` adds
# everything in the current folder
git add .
git status
git commit -m "Export clean data as CSV."
```


```bash
# Add code to export clean data...Note that `add .` adds
# everything in the current folder
git add .
git status
git commit -m "Export clean data as CSV."
```

3. Merge

Once we've developed and tested the changes on our `data_analysis` branch, we can include them in `master` by **merging** the two branches.

```bash
# Move back to master
git checkout master

# Merge changes on data_analysis with code on master
git merge data_analysis

# Delete the data_analysis branch
git branch -d data_analysis
```

N.b., deleting the `data_analysis` branch isn't necessary, but it's best practice to prune unneeded branches.

```
# Create a new directory, and initialize a Git repo inside of it.
mkdir git_practice
cd git_practice

# Create an `hello.py`. In the page body, put a heading with the text `"Welcome"`, and a paragraph with Lorem text.
touch hello.py

# Add and commit your `hello.py`.
git add hello.py
git commit -m "First commit."

# Create and checkout a new branch, called `helpers`.
git branch helpers
git checkout helpers
# Or: git checkout --branch helpers

# Add greet function to helpers.py
git add helpers.py
git commit -m "Add greet function to helpers.py."

# Update hello.py
git add hello.py
git commit -m "Refactor hello.py to use greet function."

# Move back to your `master` branch.
git checkout master

# Merge `master` with your `helpers` branch.
git merge helpers

# Delete your `helpers` branch.
git branch -d helpers
```

jupyter setUpstream.sh Last Friday at 1:19 PM

File Edit View Language

```
# Switch to master
git checkout master

# Set origin/master as the default branch to push to
# when on master
git push origin master -u

# Now, when we're on master, this is the same as:
# 'git push origin master'
git push

# Switch to push_example
git checkout push_example

# Set origin/push_example as the default branch to push to
# when on push_example
git push -u origin push_example

# Now, when we're on push_example, this is the same as:
# git push origin push_example
git push

```

```
# Create a git repository somewhere...
# ...Then, track a remote called origin to your local repo
# Skip if you've cloned a repo!
# git remote add origin <repo_url>

# Switch to master
git checkout master

# Now, push the master branch to GitHub
git push origin master

# Create and checkout a new branch
git checkout -b push_example

# Push new branch to GitHub
git push origin push_example

# Switch to master
git checkout master

# After others have pushed their branches to GitHub...
git pull

```

jupyter solution.sh Last Friday at 1:19 PM

File Edit View Language

```
# Create a new GitHub repo to associate with the local Git repo you've been working on thusfar. Add it as the `origin` remote.
git remote add origin <github_repo_url>

# Next, checkout your `master` branch, and push it to GitHub.
git checkout master
git push origin master
# Or: git push -u origin master

# Create and checkout a new branch
git branch add_gitignore
git checkout add_gitignore
# Or: git checkout -b add_gitignore

# Change something in your project-this adds a .gitignore file
echo ".DS_STORE" > .gitignore
git add .gitignore
git commit -m "Add .gitignore file."

# Push this branch to GitHub
git push -u origin add_gitignore

# Checkout `master`, and merge it with the branch you just created.
git checkout master
git merge add_gitignore

# Push the updated `master` branch to GitHub.
git push
# Or: git push origin master

```

Branching in Git

This document contains a branching workflow cheatsheet.

Branching Workflow

1. Create a branch: `git branch <branch_name>`
2. Checkout the new branch: `git checkout <branch_name>`
3. Use `git add` and `commit` as normal.
4. When you want to push your branch to GitHub, do: `git push origin <branch_name>`
5. When you want to merge your new branch into master, do the following four steps:
   1. `git checkout master`
   2. `git pull origin master`
   3. `git merge <branch_name>`

Note that you can do `git checkout --branch <branch_name>` to create a new branch _and_ immediately check it out with a single command.

- - -

Branching Recipes

Initialize Repository

To initialize a Git repo from the command line:

```
git init
```

Display All Branches

To display all branches:

```
git branch
```

Display All Branches, with Metainformation

To display all branches, including information about the latest commit on each branch.

```
git branch --verbose

# Or, as shorthand
git branch -v
```

Create a new branch

To create a new branch, but _not_ switch to it:

```
git branch <branch_name>
```

Move to an Existing Branch

To move to a branch that already exists:

```
git checkout <branch_name>
```

Create & Checkout New Branch Simultaneously

To create and immediately checkout a new branch:

```
git checkout --branch <branch_name>

# Or, as shorthand
git checkout -b <branch_name>
```

Add a Remote for Push/Pull

To add a remote to push to/pull from:

```
git remote add <remote_name> <remote_url>
```

For example, to add a GitHub repo called `remote_example` as `origin`, do:

```
git remote add origin github.com/peleke/remote_Example
```

Merge Branches

To merge a branch into the branch you're currently on:

```
git merge <branch_name>
```

For example, doing:

```
# While on `master`...
git merge new_feature
```

...Will merge the `new_feature` branch into `master`.

Push to GitHub

To push changes for a branch to GitHub:

```
git push origin <branch_name>
```

To set a default upstream branch, add the `-u` flag. For example, doing this:

```
# While on the `master` branch...
git push -u origin master
```

...Makes it such that we can just write `git push` instead of `git push origin master`.

Aliases

Aliases allow you to give your own names to various Git commands, which allows us to save typing.

```bash
# Run the following to install common aliases

git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.st status
```

This allows us to write `git ci -m "First"` instead of `git commit -m "First"`.

- - -