Commit early, commit often!

A gentle introduction to the joy of Git and GitHub

Setup Steps

Setup Step A. GitHub Account

- To actively participate in this workshop, you'll need credentials for an account with GitHub.
- If you don't already have an account, you can create one for free at https://github.com/signup

Setup Step B. GitHub Personal Access Token

- Please create a personal access token with these properties:
  - Enter Note "For WUSS 2022 HOW"
  - Set Expiration of "7 days"
  - Select only the scope "public_repo"
- Once you've created your token, it's recommended to copy/paste it into a password manager or text file on your local machine. In addition, you might want to leave the tab showing your token open.
- **Note**. Personal Access Tokens should be treated like passwords. For this reason, GitHub doesn't allow you to view a token after it's been created.

Setup Step C. Google Account (for practicing the Git CLI in Google Colab)

- If you see a **Sign In** button in the upper right corner, you'll need to sign into a Google account before you can execute code cells in this Notebook.
- If you don't already have an account, you can create one for free at https://accounts.google.com/signup

Setup Step D. Get ready to use Google Colab

- Enable line numbers in the Tools menu: **Tools** -> **Settings** -> **Editor** -> **Show line numbers** -> **Save**
- Enable the Table of Contents in the View menu: **View** -> **Table of contents**
- Save a copy of this notebook, along with any edits you make, in the File menu: **File** -> **Save a copy in Drive**
- Looking for "extra credit"? Please let us know if you spot any typos!

Part 1. Practicing the GitHub web interface

Instructions:

1. Visit https://github.com/ to make sure you're signed into a GitHub account.
2. Make a *fork* (i.e., a separate, personal copy) of https://github.com/saspy-bffs/wuss-2022-how-git-notes
3. Make a *branch* (i.e., an internal working copy) of your fork called *my-git-notes*.
4. Make a *commit* (i.e., save some changes) to the file `README.md`.
5. Open a *Pull Request* (aka a PR) within your fork.
6. Open a *Pull Request* (aka a PR) from your fork to `saspy-bffs/wuss-2022-how-git-notes`.

Recommendation: Throughout this workshop, leave the tab for your `README.md` file open, and take notes. This will help you work through all of the Git vocabulary.

Related GitHub Documentation:

- Creating a fork
- Creating a branch
- Creating a commit
- Opening a PR

Section 1 Concept Check

1. Short Answer: What are some reasons we might want to create a fork of a repo?
2. Short Answer: What are some reasons we might want to work in branches?
3. Short Answer: What are some reasons we might want to open a PR within our own repo?
4. Short Answer: What are some reasons we might want to open a PR to the repo we forked from?

Some Possible Responses to the Section 1 Concept Check

1. Short Answer: What are some reasons we might want to create a fork of a repo?

   Notes: Creating a fork allows us to maintain our own separate, personal copy of a repo, indepdently of the original repo. A fork can be used for many purposes, including personal interest, customizing open-source code, and using PRs from our fork to the original repo in order to make contributions to an open-source project.

2. Short Answer: What are some reasons we might want to work in branches?

   Notes: Working in branches allows to us keep our *main* branch as a known reference point, and to periodically merge changes into *main* once we're sure we're ready to change it. In particular, many developers find it helpful to make "exploratory changes" in branches, and to throw the branches away if something goes amiss.

3. Short Answer: What are some reasons we might want to open a PR within our own repo?

   Notes: PRs are a great way of organizing collections of commits into a single package. For example, each PR could represent a major new version of milestone being reached. The GitHub web interface makes it straightforward to view the "diffs" for PR,

which allows us to quickly see what changes are reflected in a PR.

4. Short Answer: What are some reasons we might want to open a PR to the repo we forked from?

   Notes: One of the most popular developer workflows (especially for contributing to open-source projects) is called the "Git Flow": Fork a repo, make a local clone, make a branch to work in, commit some changes, and open a PR against the original repo. In this sense, the PR becomes a request for the maintainers of the original repo to pull the changes you've made into their project.

Section 2. Practicing the Git CLI (Command Line Interface)

2.1. Trying out `git` from the Linux CLI

2.1.1. Check Google Colab's underlying OS

```
%%shell
cat /etc/*release*
```

```
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=18.04
DISTRIB_CODENAME=bionic
DISTRIB_DESCRIPTION="Ubuntu 18.04.6 LTS"
NAME="Ubuntu"
VERSION="18.04.6 LTS (Bionic Beaver)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 18.04.6 LTS"
VERSION_ID="18.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
VERSION_CODENAME=bionic
UBUNTU_CODENAME=bionic
```

Notes about Section 2.1.1.

- In this example, we've used the standard shell command to determine the Linux distribution we're using.
  - The `cat` (aka *concatenate*) command prints the contents of a file.
  - The directory `/etc` (aka et cetera) is the standard place in Linux to find system-wide configuration files.
  - Depending on the distribution of Linux we're using, a different file in `/etc` might contain OS information, which is why we're using the wildcard pattern `*release*`.
- By default, cells in Google Colab are expected to contain Python code, which is why we're including the magic command `%%shell` to override this behavior.
- The `%%shell` command tells Google Colab to interpret the contents of a cell as Linux shell commands.

2.1.2. Check the version of Git

```
1 %%shell
2 git --version
```

```
git version 2.17.1
```

Notes about Section 2.1.2.

- In this example, we've invoked the `git` command-line interface, and we've included the `--version` option. (The two hyphens preceeding the command mean we're asking `git` itself for information, rather than issuing a sub-command. We'll see many examples of sub-commands later.)
- As of this writing, Google Colab provides `git` version 2.17, whereas version 2.37 is available from https://git-scm.com/downloads.
- As before, the `%%shell` command tells Google Colab to interpret the contents of a cell as Linux shell commands.

2.1.3. View available `git` command-line options and sub-commands

```
1 %%shell
2 git --help
```

```
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add        Add file contents to the index
   mv         Move or rename a file, a directory, or a symlink
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
   bisect     Use binary search to find the commit that introduced a bug
   grep       Print lines matching a pattern
   log        Show commit logs
   show       Show various types of objects
   status     Show the working tree status

grow, mark and tweak your common history
   branch     List, create, or delete branches
   checkout   Switch branches or restore working tree files
   commit     Record changes to the repository
   diff       Show changes between commits, commit and working tree, etc
   merge      Join two or more development histories together
   rebase     Reapply commits on top of another base tip
   tag        Create, list, delete or verify a tag object signed with GPG
```

```
collaborate (see also: git help workflows)
   fetch      Download objects and refs from another repository
   pull       Fetch from and integrate with another repository or a local branch
   push       Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
```

Notes about Section 2.1.3.

- In this example, we've invoked the `git` command-line interface, and we've include the `--help` option.
- In addition to the usage syntax with options like `--version` and `--help`, you should also see groups of sub-commands listed.
- We'll be focusing on the following seven most commonly used sub-commands, listed in the order they appear in `git --help`:
    1. `git clone` to make a local copy of a remote code repository
    2. `git add` to track changes in files
    3. `git status` to check the status of changes in our code repository
    4. `git branch` to list and create branches
    5. `git checkout` to switch to a different branch
    6. `git commit` to mark file changes as part of our Git history
    7. `git push` to push committed file changes up to a remote version of our local code repository
- We'll also use `git config` to set and retrieve information about our GitHub identity.
- For more about the many `git` sub-commands, see https://git-scm.com/docs.
- For more about the most commonly used `git` sub-commands, see https://training.github.com.
- As before, the `%%shell` command tells Google Colab to interpret the contents of a cell as Linux shell commands.

2.1.4. Setup your GitHub identity

```
%%shell
git config --global user.name REPLACE_THIS_LONG_VARIABLE_NAME_WITH_YOUR_GITHUB_USERNAME
git config --global user.email REPLACE_THIS_LONG_VARIABLE_NAME_WITH_YOUR_GITHUB_EMAIL_ADDRESS
git config -l
```

```
user.name=REPLACE_THIS_LONG_VARIABLE_NAME_WITH_YOUR_GITHUB_USERNAME
user.email=REPLACE_THIS_LONG_VARIABLE_NAME_WITH_YOUR_GITHUB_EMAIL_ADDRESS
```

Notes about Section 2.1.4.

- In this example, we use the `git config` sub-command to set our global `git` identity using our GitHub username and email address.
- We then use `git config` with the `-l` option to list our current configuration settings.
- As before, the `%%shell` command tells Google Colab to interpret the contents of a cell as Linux shell commands.

2.2. Setting up a local clone of a remote Git repo

2.2.1. Look inside the local file system

```
%%shell
ls -R1sh
```

```
.:
total 4.0K
4.0K sample_data

./sample_data:
total 55M
4.0K anscombe.json
296K california_housing_test.csv
1.7M california_housing_train.csv
 18M mnist_test.csv
```

```
 35M mnist_train_small.csv
4.0K README.md
```

Notes about Section 2.2.1.

- In this example, we use the `ls` (aka *list*) command with several flags to list the contents of the current directory (denoted by a single period in Linux) and its subdirectories:
  - The `-R` flag means to recursively look in all subdirectories.
  - The `-1` flag means to list directory contents in a single column.
  - The `-s` flag means to include the sizes of all files.
  - The `-h` flag means to print file sizes in human-readable units (as opposed to the number of blocks used on disk).
- You can also view these contents by clicking on the folder icon in the left-hand panel of Colab's web interface.
- As before, the `%%shell` command tells Google Colab to interpret the contents of a cell as Linux shell commands.

2.2.2. Make a local clone of your fork of `saspy-bffs/wuss-2022-how-git-notes`

```
%%shell
export gh_username="$(git config --global user.name)"
read -s -p 'Please Enter Your GitHub Personal Access Token: ' gh_pat
echo -e '\n'
git clone https://$gh_pat@github.com/$gh_username/wuss-2022-how-git-notes.git
echo ''
ls -R1sh
```

```
Please Enter Your GitHub Personal Access Token:

Cloning into 'wuss-2022-how-git-notes'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 9 (delta 2), reused 3 (delta 0), pack-reused 0
```

```
Unpacking objects: 100% (9/9), done.

.:
total 8.0K
4.0K sample_data
4.0K wuss-2022-how-git-notes

./sample_data:
total 55M
4.0K anscombe.json
296K california_housing_test.csv
1.7M california_housing_train.csv
 18M mnist_test.csv
 35M mnist_train_small.csv
4.0K README.md

./wuss-2022-how-git-notes:
total 8.0K
4.0K LICENSE
4.0K README.md
```

Notes about Section 2.2.2.

- In this example, we use the `git clone` subcommand to make a clone (i.e., a separate, local copy) of our GitHub repo, using our Personal Access Token (PAT) to authenticate, and we then print the contents of our current working directory again to show our local clone.
- To automate this process, we first set up a couple of pieces of information:
    - We load our GitHub username into the environment variable `gh_username` using the `export` command together with Linux shell command substitution. (In other words, `gh_username` takes on the result of executing the command `git config --global user.name`, which returns our the username we set above.)
    - We use the Linux `read` command with the `-p` flag to prompt the user for their GitHub PAT and `-s` to suppress what's typed (since PATs should be treated like passwords), and we store the result in the environment variable `gh_pat`.
- Finally, we use the `echo` command to print blank lines between output components.

- You can also confirm the `wuss-2022-how-git-notes` directory was created by refreshing the Files view in the left-hand panel (e.g., by right-clicking and selecting "Refresh").
- As before, the `%%shell` command tells Google Colab to interpret the contents of a cell as Linux shell commands.

2.2.3. Navigate into your local clone of your fork of `saspy-bffs/wuss-2022-how-git-notes`

```
1 %cd wuss-2022-how-git-notes
```

```
/content/wuss-2022-how-git-notes
```

Notes about Section 2.2.3.

- In this example, we use the magic command `%cd` to have Google Colab change its current working directory, navigating into our local clone.
- Because Google Colab runs each cell prefixed with `%%shell` in an isolated sub-shell that's immediately discarded, we have to tell Google Colab to change directories.
- If we were working in a typical Linux CLI environment, we would instead use the Linux command `cd wuss-2022-how-git-notes`.

2.2.4. Look inside your local clone of your fork of `saspy-bffs/wuss-2022-how-git-notes`

```
1 %%shell
2 ls -1Ash
```

```
total 12K
4.0K .git
4.0K LICENSE
4.0K README.md
```

Notes about Section 2.2.4.

- In this example, we once again use the `ls` (aka *list*) command, but with a slightly different set of flags to list the contents of the current directory only:
  - The `-1` flag means to list directory contents in a single column.
  - The `-A` flag means to list all contents, including files/folders starting with a period (`.`). In Linux, a file or folder will typically have a period at the start of its filename if it's used for configuration. Here, the *.git* directory holds information about our code repository.
  - The `-s` flag means to include the sizes of all files.
  - The `-h` flag means to print file sizes in human-readable units (as opposed to the number of blocks used on disk).
- As before, the `%%shell` command tells Google Colab to interpret the contents of a cell as Linux shell commands.

2.3. Working with Branches

2.3.1. Start with the command you'll use most often

```
1 %%shell
2 git status
```

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

nothing to commit, working tree clean
```

Notes about Section 2.3.1.

- In this example, we use the `git status` subcommand to get information about the repo we're currently in.
- When working with the `git` CLI, it's common to use `git status` frequently for information about file changes and which branch we're currently working in, as we'll see below.

- Assuming the cells in this Notebook are being run consecutively with all instructions followed in order, we should be told we have a clean working tree, which is always a happy message to see.
- The name *origin/main* means the remote version of the branch *main*, which is special. Typically, we'll want to work inside a branch of our local clone and only merge changes into *main* periodically, with the *main* branch used as a reference point of known/working code.
- As before, the `%%shell` command tells Google Colab to interpret the contents of a cell as Linux shell commands.

2.3.2. List all of the branches available to us

```
1 %%shell
2 git branch -a
```

```
  * main
    remotes/origin/HEAD -> origin/main
    remotes/origin/main
    remotes/origin/my-git-notes
```

Notes about Section 2.3.2.

- In this example, we use the `git branch` subcommand with the `-a` flag to list all of the branches of our repo, available both locally and remotely.
- Assuming the cells in this Notebook are being run consecutively with all instructions followed in order, there should be four branches listed:
    1. The local *main* branch, in green with an asterisk preceding it, since it's the local branch we're currently working in.
    2. The remote *HEAD* branch, which is used to name the default remote branch (here, *origin/main*).
    3. The remote *main* branch (aka *origin/main*), with *origin* indicating that we made a local clone of a remote repo.
    4. The remote *my-git-notes* branch, which you were asked to create in Part 1.

- The three actual branches listed (*main*, *origin/main*, and *origin/my-git-notes*) are effectively three separate copies of the files in our repo, which we can edit independently before using git commands to sync their contents.
- As before, the `%%shell` command tells Google Colab to interpret the contents of a cell as Linux shell commands.

2.3.3. Make a new local branch

```
%%shell
git branch local-test-branch
echo ''
git branch -a
echo ''
git status
```

```
  local-test-branch
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
  remotes/origin/my-git-notes

On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

Notes about Section 2.3.3.

- In this example, we use the `git branch` subcommand without any flags to create a new local branch.
- We then repeat two sub-commands we've seen before (`git branch -a` and `git status`) to list our branches and get repo status. Note that the local branch we're currently working in hasn't changed. (To create a local branch and instantly switch to it, we could have used the sub-command `git checkout -b <branch-name>` instead.)

- Assuming the cells in this Notebook are being run consecutively with all instructions followed in order, there should be five branch listed, representing four separate copies of the files in our repo (*local-test-branch*, *main*, *origin/main*, and *origin/my-git-notes*).
- Typically, when we work under version control, we like to work inside branches. As mentioned above, this allows us to keep *main* as a known, working reference point that we will only change periodically using Pull Requests (PRs).
- As before, the `echo` command prints blank lines between output components, and the `%%shell` command tells Google Colab to interpret the contents of a cell as Linux shell commands

2.3.4. Switch to our new branch

```
1 %%shell
2 git checkout local-test-branch
3 echo ''
4 git branch -a
```

```
Switched to branch 'local-test-branch'

* local-test-branch
  main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
  remotes/origin/my-git-notes
```

Notes about Section 2.3.4.

- In this example, we use the `git checkout` subcommand to change the local branch we're currently working in.
- We then repeat the `git branch -a` sub-command to list our branches.
- As before, the `echo` command prints blank lines between output components, and the `%%shell` command tells Google Colab to interpret the contents of a cell as Linux shell commands.

2.3.5. Try switching to a remote branch

```
1 %%shell
2 git checkout my-git-notes
3 echo ''
4 git branch -a
```

```
Branch 'my-git-notes' set up to track remote branch 'my-git-notes' from 'origin'.
Switched to a new branch 'my-git-notes'

  local-test-branch
  main
* my-git-notes
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
  remotes/origin/my-git-notes
```

Notes about Section 2.3.5.

- In this example, we once again use the `git checkout` subcommand to change the branch we're currently working in.
- This time, though, because there's only a remote branch named *my-git-notes*, git first makes a local copy of the remote branch before switching branches.
- Assuming the cells in this Notebook are being run consecutively with all instructions followed in order, there should be six branch listed, representing five separate copies of the files in our repo (*local-test-branch*, *main*, *my-git-notes*, *origin/main*, and *origin/my-git-notes*).
- As before, the `echo` command prints blank lines between output components, and the `%%shell` command tells Google Colab to interpret the contents of a cell as Linux shell commands.

2.4. Working with Local Changes

2.4.1. Create a new local file

```
1 %%shell
```

```
2 touch my-new-local-file.txt
3 ls -1Ash
4 echo ''
5 git status
```

```
total 12K
4.0K .git
4.0K LICENSE
   0 my-new-local-file.txt
4.0K README.md

On branch my-git-notes
Your branch is up to date with 'origin/my-git-notes'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        my-new-local-file.txt

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

Notes about Section 2.4.1.

- In this example, we use the `touch` command to create a new file named *my-new-local-file.txt*. (In general, `touch` updates the last-modified date of a file, creating the file if it doesn't already exist.)
- We then repeat two commands we've seen before (`ls -1Ash` and `git status`) to list the files in our repo and get repo status.
- Assuming the cells in this Notebook are being run consecutively with all instructions followed in order, the output of `git status` should show that changes have been made to one untracked file in our current working branch.
- As before, the `echo` command prints blank lines between output components, and the `%%shell` command tells Google Colab to interpret the contents of a cell as Linux shell commands.

2.4.2. Tell `git` to track changes to our new file

```
1 %%shell
```

```
2 git add my-new-local-file.txt
3 echo ''
4 git status
```

```
On branch my-git-notes
Your branch is up to date with 'origin/my-git-notes'.

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

        new file:   my-new-local-file.txt
```

Notes about Section 2.4.2.

- In this example, we use the `git add` subcommand to add *my-new-local-file.txt* to the tracked-files list.
- We then repeat the `git status` sub-command to get repo status.
- Assuming the cells in this Notebook are being run consecutively with all instructions followed in order, the output of `git status` should show that changes have been made to one tracked file in our current working branch.
- Git marks changed files as "Untracked" by default in order to give us greater flexibility. For example, we might edit 10 files in a directory, but only want to track changes in 7 of them.
- As before, the `echo` command prints blank lines between output components, and the `%%shell` command tells Google Colab to interpret the contents of a cell as Linux shell commands.

2.4.3. Tell `git` to commit these changes to our Git History

```
1 %%shell
2 git commit -m "Create my-new-local-file.txt"
3 echo ''
4 git status
```

```
[my-git-notes a2eda44] Create my-new-local-file.txt
```

```
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 my-new-local-file.txt

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

nothing to commit, working tree clean
```

Notes about Section 2.4.3.

- In this example, we use the `git commit` subcommand with the `-m` flag to create a commit with commit message "Create my-new-local-file.txt".
- In other words, we've just created a discrete, named point in the history of the changes to the files in our repo, and we've used the name "Create my-new-local-file.txt"
- When writing commit messages, it's considered good practice to use the imperative mood, as if you were giving an instruction to a colleague for the exact change they should make, in 50 characters or less.
- This allows us to look through the list of all commit messages (viewable using `git log`) and identify a specific point in a repo history, should we need to look back at it for reference. (The state of a repo corresponding to a specific commit can be viewed with the `git checkout` command.)
- Earlier, we saw the *.git* folder, which is where the changes for each of our commits is tracked.
- Assuming the cells in this Notebook are being run consecutively with all instructions followed in order, the output of `git status` should show that our local working tree is clean, but that our chances don't yet appear in the remote GitHub repo since our local clone is ahead by one commit.
- As before, the `echo` command prints blank lines between output components, and the `%%shell` command tells Google Colab to interpret the contents of a cell as Linux shell commands.

2.4.4. Tell `git` to push our local changes back to GitHub

```
1 %%shell
2 git push
```

```
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 331 bytes | 331.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/ilankham/wuss-2022-how-git-notes.git
   3c8180e..a2eda44  my-git-notes -> my-git-notes
```

Notes about Section 2.4.4.

- In this example, we use the `git push` command to publish our local changes to our remote GitHub repo.
- By clicking on the URL in the output, you'll be taken to the landing page for your repo, where you can see the results of your push and initial a Pull Request (aka PR).
- A common workflow for a team using GitHub collaboratively is as follows:
    1. The team creates the repo in the GitHub web interface.
    2. Each teammate makes a local clone.
    3. In their local clone, each teammate works in a branch (created either locally or as a copy of a pre-existing remote branch).
    4. As a teammate makes edits in their local working branch, they use `git add` to track files, as needed.
    5. Then, when a teammate is ready to publish a collection of changes to one or more tracked files, they use `git commit`.
    6. The commit is then pushed to the remote GitHub repo (with `git push`), and a PR is opened (e.g., using the "Compare & pull request" button on the repo landing page).
    7. Using the GitHub web interface, other team members can then visually inspect the contents of the PR.
    8. The GitHub web interface can also be used to annotate code within a PR, as well as to have entire conversations.
    9. If additional changes are committed to the branch the PR was opened against and then pushed to GitHub, they will be added to the Pull Request.
    10. Then, once the team is satisified with the exact changes in the PR, the PR can be merged into the main branch.
- If we wish to pull in changes made remotely, we can also use the companion `git pull` sub-command.
- As before, the `%%shell` command tells Google Colab to interpret the contents of a cell as Linux shell commands.

Part 3. Open Lab Time

Instructions:

1. Work through the GitHub Hello World, taking notes in your `wuss-2022-how-git-notes` repo as you do.
2. Then spend some time making additional changes in branches and using Pull Requests (aka PRs) to merge them into *main*.
3. In particular, you should try the following:
    - Open a PR.
    - Make a new commit in the branch the PR was based on.
    - Go back to the webpage for the PR to see the new commit added.

      This behavior is especially important since it means a PR doesn't need to be closed and reopened to modify its change set.

Notes and Resources

Want some ideas for what to do next? Here are our suggestions:

1. Try repeating all of the steps of this HOW outside of Google Colab. Here are two recommended options:
    - Repeat all of the steps of this HOW using a local installation of `git`.

      In you're using Linux or a Unix-like operating system (e.g., macOS), all of the shell commands above should work in a terminal.

      If you're using windows, the shell commands can be used in the Git Bash terminal, which is installed with Git.
    - Repeat all of the steps of this HOW using a GUI client like GitHub Desktop, which allows many of the same operations to be carried out using a point-and-click interface.

2. Try a phrase like "git tutorial" in your favorite search engine. You'll find many different types of tutorials taking different tones, and sometimes providing interactive examples. Git may feel convoluted at times, but it's ubiquitous. Many software developers consider it a rite of passage to make a personal Git tutorial, hence the large variety fitting different ways of thinking about and using Git.
3. Keep in touch for follow-up questions/discussion (one of our favorite parts of teaching!) using isaiah.lankham@gmail.com and matthew.t.slaughter@gmail.com
4. You can also use your GitHub account to chat with us on Gitter at https://gitter.im/saspy-bffs/community