Commit early, commit often!

A gentle introduction to the joy of Git and GitHub

A few notes before we get started...

Google Colab Setup

- To save a copy of this notebook, along with any notes/edits you make, use the File menu: **File** -> **Save a copy in Drive**
- To enable line numbers, use the Tools menu: **Tools** -> **Settings** -> **Editor** -> **Show line numbers** -> **Save**
- To disable Gemini, use the Tools menu: **Tools** -> **Settings** -> **AI Assistance** -> **Hide generative AI features** -> **Close**
- To enable the Table of Contents, use the View menu: **View** -> **Table of contents**

Google Accounts

- If you see a **Sign In** button in the upper right corner, we recommend signing into a Google account. This will enable you to make a copy of this notebook (e.g., to take notes during the workshop), as well as to execute code examples in Section 4.
- If you don't already have a Google account, you can create one for free at https://accounts.google.com/signup

Section 1. Getting Started with GitHub

1.1. Sign into a GitHub Account

- If you already have a GitHub account, visit https://github.com/ and sign in.
- If you don't already have an account, follow the instructions at https://github.com/signup to create one for free.

1.2. Explore the GitHub Interface for Code Repositories (aka repos)

- Visit the GitHub repo for this workshop at https://github.com/saspy-bffs/wuss-2024-git-how and note the following file components:
  - A folder with a PDF of the workshop slides
  - A plain-text file with LICENSE info, which contains an open source license for the repo allowing anyone to reuse the contents freely
  - A plain-text file called README.md, which is automatically displayed below the file list (Note: This file is written in a markup language called GitHub Flavored Markdown.)
- As an example of a much larger repo being actively used by many collaborators, visit the official GitHub repo for the Python programming language at https://github.com/python/cpython and note the following collaboration elements:
  - Project management components, including the issues tracker and the pull requests tracker at the top left
  - Social media components, include the "Watch" and "Star" buttons at the top right

1.3. Explore the GitHub Interface for a User's Profile

- Click the profile icon in the upper-righthand corner, select **Your profile**, and note the following:
  - This is a public landing page with an overview of your GitHub activity, which is often used as a form of résumé in the software-engineering world.
  - There are also gamification elements, like achievements and the contribution graph.
  - If you want to bookmark a single page for GitHub, this is a good option.
- Click the profile icon in the upper-righthand corner, select **Settings**, and note the following:
  - There are numerous options for customizing your public landing page, and your GitHub identity in general.
  - At the bottom of the lefthand navigation menu, there's a section called **Developer settings**, which is where a "Personal Access Token" for Section 4 below can be created.

1.4. Related GitHub Documentation

- Signing up for a new GitHub account
- About your profile
- Personalizing your profile
- Licensing a repository
- Adding a file to a repository
- GitHub Flavored Markdown Spec

Section 2. Working with Repos in GitHub

2.1. Practice Workflow 1: Commits in "Main Branch"

- Review Slide 5 in the HOW Slides
- Follow these instructions to make a new public repo and add one or more commits to the "main branch" of the repo: https://docs.github.com/en/get-started/quickstart/create-a-repo
- In the upper-righthand corner of the file view for your repo, locate the clock icon with text like "5 commits" next to it. (Here, "5" is the total number of commits that have been made to the repo. The number of commits for your repo may be different.)
- Click on the clock icon to see the full commit history for the repo.
- Click on one of the commit messages to see the full change history for the commit, and then click on the **Browse files** button near the upper-righthand corner to look through the repo files at that point in the repo history.
- To return to the current state of the repo, click on the repo name in the upper-lefthand corner.

2.2. Practice Workflow 2: Commits in "Feature Branch" + Pull Request (aka PR)

- Review Slide 6 in HOW Slides
- Follow these instructions to make a new repo and complete the full "Hello World" exercise: https://docs.github.com/en/get-started/quickstart/hello-world
- Note the following benefits over Workflow 1 (Commits in "Main Branch"):
  - Exploratory Coding: Working in a "feature branch" is like working in a "parallel universe." We can throw away a feature branch without changing our main branch, or we can merge changes "back into main" whenever we're ready. Either way, we can keep the main branch as a fixed reference point while we work in our feature branch.
  - Simultaneous Collaboration: Collaborators can work in separate feature branches, allowing them to make changes to the same files and rely on Git to merge their changes together intelligently.
  - Differentiated permissions: Some collaborators may only need permission to open pull requests (PRs) from their feature branch to the main branch, but may not need permission to merge PRs themselves.

2.3. Related GitHub Documentation

- Creating a new repository
- Creating and deleting branches within your repository
- About commits
- Creating a pull request
- GitHub Flavored Markdown: Basic writing and formatting syntax

Section 3. Working with Forks of Repos in GitHub

3.1. Fork + Commits in "Feature Branch" + Pull Request (aka PR)

- Review Slide 7 in HOW Slides
- Visit https://github.com/saspy-bffs/wuss-2024-git-how-practice and use the "fork" button in the upper-righthand corner to create your own personal fork of the repo.
- As in Workflow 2, create a feature branch called `my-git-notes` within your fork, and make one or more commits to the file `README.md` within this feature branch.
- As in Workflow 2, open a PR, but have the PR instead use your feature branch as the "compare branch" and the main branch of the original repo as the "base branch".
- Note the following difference from Workflow 2 (Commits in "Feature Branch" + PR):
  - Workflow 2 allows us to work within a single code repository, where we have permissions to make a new branch and open a PR within the repo.
  - Workflow 3, however, is for proposing changes to a repo that we may not have permission to modify.
  - Workflow 3 is commonly used to contribute to open-source projects on GitHub, even if just to propose fixing a typo in a `README.md` file!

3.2. Related GitHub Documentation

- Forking a repository
- Creating and deleting branches within your repository
- About commits
- Creating a pull request from a fork
- GitHub Flavored Markdown: Basic writing and formatting syntax

Section 4. Working with Local Clones of Forks of Repos in GitHub

4.1. Create a GitHub Personal Access Token

- Review Slide 8 in HOW Slides
- Please create a personal access token (classic) with these properties:
  - Enter Note "For WUSS 2024 Hands-on Workshop"
  - 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 until you're sure you've saved a local copy.
- **Note**: Personal Access Tokens should be treated like passwords. For this reason, GitHub doesn't allow a token to be viewed after it's been created.

4.2. Trying out `git` from the Linux CLI

4.2.1. Check Google Colab's underlying OS

In [ ]:
```
%%shell
cat /etc/*release*
```

Notes

- In this example, we've used a standard shell command to determine the Linux distribution being used by Google Colab:
  - 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 to be handled by the underlying operating system.

4.2.2. Check the version of Git

In [ ]:
```
%%shell
git --version
```

Notes

- 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 2.34.1, whereas version 2.46.0 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 to be handled by the underlying operating system.

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

In [ ]:
```
%%shell
git --help
```

Notes

- 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 most commonly used `git` sub-commands, see the GitHub git cheat sheet.
- For a complete reference to `git` sub-commands, see https://git-scm.com/docs.
- As before, the `%%shell` command tells Google Colab to interpret the contents of a cell as Linux shell commands to be handled by the underlying operating system.

4.2.4. Setup your GitHub identity

In [ ]:
```
%%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
```

Fill in your GitHub username and email on lines 2 & 3 above, e.g.:

```
git config --global user.name ilankham
git config --global user.email isaiah.lankham@gmail.com
```

Notes

- 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 to be handled by the underlying operating system.

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

4.3.1. Look inside the local file system

In [ ]:
```
%%shell
ls -R1sh
```

Notes

- 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 to be handled by the underlying operating system.

4.3.2. Make a local clone of your fork of `saspy-bffs/wuss-2023-class-git-practice`

In [ ]:
```
%%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_username:$gh_pat@github.com/$gh_username/wuss-2023-class-git-practice.git
echo ''
ls -R1sh
```

In [ ]:
```
%%shell
git config -l
```

Notes

- 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-2023-class-git-practice` 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 to be handled by the underlying operating system.

4.3.3. Navigate into your local clone of your fork of `saspy-bffs/wuss-2023-class-git-practice`

In [ ]:
```
%cd wuss-2023-class-git-practice
```

Notes

- 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-2023-class-git-practice`.

4.3.4. Look inside your local clone of your fork of `saspy-bffs/wuss-2023-class-git-practice`

In [ ]:
```
%%shell
ls -1Ash
```

Notes

- 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 to be handled by the underlying operating system.

4.4. Working with Branches

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

In [ ]:
```
%%shell
git status
```

Notes

- 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 to be handled by the underlying operating system.

4.4.2. List all of the branches available to us

In [ ]:
```
%%shell
git branch -a
```

Notes

- 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 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 Section 3.
- 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 to be handled by the underlying operating system.

4.4.3. Make a new local branch

In [ ]:
```
%%shell
git branch local-test-branch
echo ''
git branch -a
echo ''
git status
```

Notes

- 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 to be handled by the underlying operating system.

4.4.4. Switch to our new branch

In [ ]:
```
%%shell
git checkout local-test-branch
echo ''
git branch -a
```

Notes

- 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 to be handled by the underlying operating system.

4.4.5. Try switching to a remote branch

In [ ]:
```
%%shell
git checkout my-git-notes
echo ''
git branch -a
```

Notes

- 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 to be handled by the underlying operating system.

4.5. Working with Local Changes

4.5.1. Create a new local file

In [ ]:
```
%%shell
touch my-new-local-file.txt
ls -1Ash
echo ''
git status
```

Notes

- 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 to be handled by the underlying operating system.

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

In [ ]:
```
%%shell
git add my-new-local-file.txt
echo ''
git status
```

Notes

- 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 to be handled by the underlying operating system.

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

In [ ]:
```
%%shell
git commit -m "Create my-new-local-file.txt"
echo ''
git status
```

Notes

- 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 changes 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 to be handled by the underlying operating system.

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

In [ ]:
```
%%shell
git push
```

Notes

- 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 to be handled by the underlying operating system.

Notes and Resources

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

1. Try repeating Section 4 outside of Google Colab using both of these options:

   - Use 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 after git has been installed.

     - If you're using Windows, the shell commands can be used in the Git Bash terminal, which is installed alongside Git.

   - Use a GUI client like GitHub Desktop, which allows many of the same operations to be carried out using a point-and-click interface.

   Because GitHub Desktop is essentially executing Git commands in a hidden terminal session, it's recommended to learn the underlying Git commands first so that you have a sense for what GitHub Desktop is actually doing behind the scenes.

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