# Princeton University
# COS 333: Advanced Programming Techniques
# Git and GitHub Primer

## Introduction

*Git* is a distributed version control system. It was created by Linus Torvalds – the same person who created the Linux operating system – in 2005 for the purpose of managing multi-programmer development of the Linux kernel. Using Git you create repositories containing, typically, source code.

*GitHub* is an Internet service for hosting Git repositories. It was created in 2007 by Chris Wenstrath, P. J. Hyett, Tom Preston-Werner, and Scott Chacon. It's an Internet service for hosting Git repositories.

You must use Git and GitHub for your project. You should use Git and GitHub for your COS 333 *assignments* too.

This document describes a subset of Git and GitHub that may be sufficient for your work in COS 333. The first sections of this document describe setup steps that you should perform near the beginning of the semester. The remaining sections describe common use cases and workflow.

## Let's say that you want to use Git and GitHub for Assignment 1...

## Setup Step 1: Installing Git

If you haven't already installed Git, then perform this step one time only, near the beginning of the semester.

Install the Git program on your *local computer*, that is, computer that you will use to do software development. The instructions for installing Git depend upon what kind of local computer you will use. If your local computer is:

- A Mac computer (running OS X version 10.9 or above), then issue a `git` command in a terminal window. If Git isn't already installed, then OS X will prompt you to install the Xcode command-line tools. Installing the Xcode command-line tools also installs Git.
- A computer running Microsoft Windows, then browse to http://git-scm.com/download/win. The download and install will start automatically.
- A computer running Linux, then use your package manager to install Git.

## Setup Step 2: Configuring Git

If you haven't already configured Git, then perform this step one time only, near the beginning of the semester.

Configure Git to indicate your identity and preferences. To do that, issue these commands in a terminal window on your local computer:

```
$ git config --global user.name "yourname"
$ git config --global user.email youremailaddress
$ git config --global color.ui auto
$ git config --global core.editor yourpreferrededitor
$ git config --global pull.rebase false
```

**Example**: I might issue these commands:

```
$ git config --global user.name "Robert Dondero"
$ git config --global user.email rdondero@princeton.edu
$ git config --global color.ui auto
$ git config --global core.editor emacs
$ git config --global pull.rebase false
```

For *youremailaddress* I recommend that you specify your Princeton email address, but it's fine to specify any of your email addresses.

Git uses *yourpreferrededitor* if you issue a `git commit` command without the `-m` option. The `git commit` command is described later in this document.

Setting `pull.rebase` to `false` makes the command `git pull` identical to the command `git fetch` followed by the command `git merge`, as you probably want. Those commands are shown later in this document.

**Suggestion for MS Windows users**: As you may know, MS Windows computers represent each end-of-line mark as CRLF, that is, as a CR character ('\d' character, alias 0D (hex) character) followed by a LF character (alias '\n' character, alias 0A (hex) character). In contrast, Unix-based computers such as Macs and Linux computers represent each end-of-line mark as LF only. If you use a MS Windows computer and your teammate does not, then you and your teammate may experience "line-ending churn," in which Git sees an entire file as modified simply because the end-of-line marks differ. If your computer is a MS Windows computer and your teammate's computer is *not* a MS Windows computer, then issue this command:

```
$ git config --global core.autocrlf true
```

Having issued that command, when you commit a file from your working directory to your local repository Git automatically changes any CRLF end-of-line marks that occur in your working directory file to LF end-of-line marks in your local repository file. And when you check a file out of your local repository to your working directory, Git automatically changes any LF end-of-line marks that occur in your local repository file to CRLF end-of-line marks in your working directory file.

In response to those commands Git stores your settings in a file named `.gitconfig` in your home directory on your local computer.

# Setup Step 3: Creating a GitHub Account

If you haven't already created a GitHub account, then perform this step one time only, near the beginning of the semester.

Create a GitHub free account. Doing so will allow you to create private GitHub repositories with an unlimited number of collaborators. To do that, browse to https://github.com. Click on the *Sign up for GitHub* button, and follow the instructions.

# Setup Step 4: Creating a Personal Access Token

If you haven't already created a personal access token, then perform this step one time only, near the beginning of

the semester.

To use GitHub via its CLI (command-line interface), you must create a GitHub *personal access token* (*PAT*). The instructions for doing so are at this page:

https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/creating-a-personal-access-token

In summary:

- Browse to https://github.com and log in.
- Click the profile icon at the upper right to display a drop-down menu.
- In the drop-down menu click on *Settings*
- In the resulting page, click on *Developer Settings*.
- In the resulting page, click on *Personal access tokens*.
- In the resulting page, click on *Tokens (classic)*.
- In the resulting page, click on *Generate new token*.
- In the drop-down, click on *Generate new token (classic)*.
- In the *Note* area, enter "COS 333". Check the *repo* checkbox. Click on the *Generate Token* button.

The resulting page displays a PAT. It's important to remember your PAT. Copy it to your computer's clipboard, and maybe store it in a file, at least temporarily.

**Shortcut**: No doubt you will find it inconvenient to remember your PAT. You also will find it inconvenient to enter your GitHub username and PAT each time you issue a `git` command that interacts with GitHub. To avoid the need to do so, issue this command one time:

```
git config --global credential.helper store
```

In response to that command, Git updates the `.gitconfig` file in your home directory, and stores your GitHub credentials in a file named `.git-credentials` in your home directory. After issuing that command you may need to enter your GitHub username and personal access token one time when interacting with GitHub. Thereafter, Git automatically uses your stored GitHub username and PAT when interacting with GitHub.

# Setup Step 5: Creating a GitHub Repository

Perform this step one time only, near the beginning of your work on Assignment 1.

Perform these steps:

- Browse to https://github.com and log in.
- Click on the *New* button.
- In the resulting page, for *Repository name* enter `cos333asgt1`, select the *Private* radio button, check the *Add a README file* checkbox, and click on the *Create repository* button.
- In the resulting COS333asgt1 page, click on the *Setting* tab, click on *Manage access*, click on the *Invite a collaborator* button, and complete the dialog to invite your teammate to collaborate.

**IMPORTANT**: Your GitHub **assignment** repositories **must be private** such that only you and your assignment teammate have access to them.

**IMPORTANT**: I prefer that your GitHub **project** repository be private such that only you, your project teammates, and the COS 333 instructors have access to it. But it's OK to make your GitHub project repository publicly readable if you have a compelling reason to do so.

# Setup Step 6: Cloning Your GitHub Repository

Perform this step one time only, near the beginning of your work on Assignment 1.

- Open a terminal window on your local computer. Issue `cd` commands to change your working directory to the one where you want your local Git repository to reside. Issue this command:

```
git clone https://github.com/yourgithubusername/cos333asgt1.git
```

**Example**: I might issue this command:

```
$ git clone https://github.com/rdondero/cos333asgt1.git
```

When prompted for a username, enter your GitHub username. When prompted for a password, enter the PAT that you generated in a previous step (or see the shortcut described previously). The command generates output similar to this:

```
Cloning into 'cos333asgt1'...
Username for 'https://github.com': yourgithubusername
Password for 'https://yourgithubusername@github.com':
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
```

Your working directory contains a new directory named `cos333asgt1` which contains a `README.md` file and a `.git` directory. The `cos333asgt1` directory is your *local repository*.

**Suggestion**: Create a file named `.gitignore` in your local repository. In your `.gitignore` file, add one file name or directory name per line. Git ignores (does not track) the named files/directories.

The file/directory names in your `.gitignore` file can contain wildcard characters, thereby specifying multiple matching files/directories. The Git online documentation describes the wildcards.

**Example**: In your working directory you might create a `.env` file that contains the definitions of secret environment variables that are used by your application. For safety you wouldn't want your `.env` file to be present in your GitHub repository. So you wouldn't want Git to track your `.env` file in your local repository either. And so it would be common to add `.env` to your `.gitignore` file.

**Example**: When Python compiles `.py` files, it often creates `.pyc` files in a `__pycache__` subdirectory that is subordinate to your working directory. For efficiency you wouldn't want your `__pycache__` directory to be present in your local or GitHub repositories. After all, the `.pyc` files are derived from your `.py` files, and need not be backed up or shared with teammates. And so it would be common to add `__pycache__` to your `.gitignore` file.

# Simple Workflows

Having completed the setup steps, you now have your own private *GitHub repository* in the GitHub cloud and your own *local repository* on your local computer. Use those repositories as vehicles for learning about Git. In particular, try implementing the use cases given in this document. Experiment!!!

# Simple Workflow 1: Adding Files

Perform this step repeatedly throughout your work on Assignment 1 as required.

Add *file1* and *file2* to your local repository and your GitHub repository. To do that, in a terminal window on your local computer issue these commands:

```
# cd to the local repo directory.
cd cos333asgt1

# Pull the GitHub repo to the local repo.
git pull
```

Use an editor to create *file1* and *file2*.

```
# Stage all changes.
git add -A

# Commit the staged changes to the local repo.
git commit -m "Message describing the commit"
```

Note: If you omit the `-m "Message describing the commit"` option, then Git launches the editor that you specified in Setup Step 2, with the expectation that you will compose a message using that editor.

```
# Push the local repo to the GitHub repo.
git push origin main

# (Optionally) check status.
git status
```

# Simple Workflow 2: Changing Files

Perform this step repeatedly throughout your work on Assignment 1 as required.

Change *file1* and *file2* in your local repository and your GitHub repository. To do that, in a terminal window on your local computer issue these commands:

```
# cd to the local repo directory.
cd cos333asgt1

# Pull the GitHub repo to the local repo.
git pull
```

Use an editor to change *file1* and *file2*.

```
# Stage all changes.
git add -A

# Commit the changes to the local repo.
git commit -m "Message describing the commit"

# Push the local repo to the GitHub repo.
git push origin main

# (Optionally) check status.
git status
```

# Simple Workflow 3: Removing Files

Perform this step repeatedly throughout your work on Assignment 1 as required.

Remove *file1* and *file2* from your local repository and your GitHub repository. To do that, in a terminal window on your local computer issue these commands:

```
# cd to the local repo directory.
cd cos333asgt1

# Pull the GitHub repo to the local repo.
git pull

# Delete file1 and file2 from your working directory.
rm file1 file2

# Stage those changes.
git add -A

# Commit the changes to the local repo.
git commit -m "Message describing the commit"

# Push the local repo to the GitHub repo.
git push origin main

# (Optionally) check status.
git status
```

# Simple Workflow 4: Resolving Conflicts

Perform this step repeatedly throughout your work on Assignment 1 as required.

A *conflict* occurs when (1) your teammate edits a file and pushes it to your GitHub repository, (2) you edit the same part of the same file differently in your local repository, and (3) you attempt to pull the file from the GitHub repository to your local repository or push the file from your local repository to the GitHub repository.

This sequence of commands illustrates a conflict and how to resolve it...

Your teammate does this:

```
# cd to the local repo directory.
cd cos333asgt1

# Pull the GitHub repo to the local repo.
git pull
```

Use an editor to change file1.

```
# Stage all changes.
git add -A

# Commit the changes to the local repo.
git commit -m "Message describing the commit"
```

You do this:

```
# cd to the local repo directory.
cd repodir

# Pull the GitHub repo to the local repo.
git pull
```

Use an editor to change *file1* such that both you and your teammate have changed the same lines of *file1*.

```
# Stage all changes.
git add -A

# Commit the changes to the local repo.
git commit -m "Message describing the commit"
```

Your teammate does this:

```
# Push the local repo to the GitHub repo.
git push origin main
```

You do this:

```
# Push the local repo to the GitHub repo; fails!
git push origin main
```

Your `git push` command fails because of a conflict. Git recommends that you pull from the GitHub repository.

```
# Pull the GitHub repo to the local repo.
git pull
```

Git notes that *file1* is in conflict. Git annotates *file1* to note the points of conflict. You edit *file1* to resolve the conflicts and eliminate the annotations.

```
# Stage all changes.
git add -A

# Commit the changes to the local repo.
```

```
git commit -m "Message describing the commit"

# Push the local repo to the GitHub repo; succeeds!
git push origin main
```

## Simple Workflow 5: Branching

Perform this step repeatedly throughout your work on Assignment 1 as required.

You decide to implement a new experimental feature in your application. It may not work out, so you don't want to implement the experiment in the `main` branch. Instead you decide to implement the experiment in a new branch named `exp`.

You issue these commands:

```
# cd to the local repo directory.
cd cos333asgt1

# Pull the GitHub repo to the local repo.
git pull

# Create a new branch named exp.
git branch exp

# Make exp the current branch.
git checkout exp
```

You edit *file1* extensively, but not completely. Oh no! Your teammate informs you of a bug in *file1* that you must fix in a hurry. It's a good thing that you're implementing your experiment in a non-`main` branch! You issue these commands:

```
# Save a temporary copy of your working directory files.
git stash

# Make main the current branch.
git checkout main
```

You edit *file1* to fix the bug. Then you issue these commands:

```
# Stage the changes.
git add -A

# Commit the changes to the local repo.
git commit -m "Message describing the commit"

# Push the local main branch to the GitHub (origin) repo.
git push origin main
```

You decide to resume your work on the experiment. Of course, you must merge your bug fix into your *exp* branch. You issue these commands:

```
# Make exp the current branch.
```

```
git stash pop

# Merge the main branch into the current (exp) branch.
git merge main
```

Git might discover that *file1* is in conflict. If so, then Git annotates *file1* to note the points of conflict. You edit *file1* to resolve the conflicts and eliminate the annotations. Then you issue these commands:

```
# Stage the changes.
git add -A

# Commit the staged files to the local repo.
git commit -m "Message describing the commit"
```

You finish editing *file1* with your experimental code. Then you issue these commands:

```
# Stage the changes.
git add -A

# Commit the changes to the local repo.
git commit -m "Message describing the commit"
```

You're happy with the outcome of the experiment, so you decide to merge your *exp* branch into the main branch. You issue these commands:

```
# Make main the current branch.
git checkout main

# Merge the exp branch into the current (main) branch.
git merge exp

# Push the main branch to the origin (GitHub) repo.
git push origin main
```

You decide that you no longer need your *exp* branch, so you issue this command:

```
# Delete the exp branch from your local repo.
git branch -d exp
```

# Realistic Workflow

Perform this step repeatedly throughout your work on Assignment 1 as required.

It's important to understand the preceding simple workflows. However, they're not realistic in typical multi-programmer projects. This section describes a more realistic workflow.

First, some important information about Git… In your local Git repository, Git maintains both ordinary branches and *remote-tracking* branches. A remote-tracking branch is a reference to the state of a remote branch, that is, to the state of a branch that exists in the GitHub repository. Quoting the *Pro Git* book, "Think of them [remote-tracking branches] as bookmarks, to remind you where the branches in your remote repositories were the last time you connected to them." An ordinary branch might have the name `main` or `newfeature`. The corresponding remote-tracking branch might have the name `origin/main` or `origin/newfeature`.

Now, let's say you intend to develop a new feature of your application. **You should not develop the new feature directly in the `main` branch**. Instead you should **develop the new feature in a new branch**, push your new branch to GitHub, on GitHub ask your teammate(s) to comment on your new branch, eventually on GitHub merge your new branch into the `main` branch, and pull the revised main branch from GitHub. These are the detailed steps:

(1) Prior to branching, make sure the local `main` branch is up to date.

```
# Update the origin/* remote-tracking branches.
git fetch origin

# Make main the current branch.
checkout main

# If necessary, merge the origin/main branch into the main branch.
# You haven't changed your main branch since the last push, so there
# should be no conflicts.
git merge origin/main
```

(2) Create a branch of the local `main` branch named `newfeature`.

```
# Create a branch named newfeature.
git branch newfeature

# Make newfeature the current branch.
git checkout newfeature
```

Use an editor to add/delete/change files in the `newfeature` branch.

```
# Stage all changes.
git add -A

# Commit the changes to the newfeature branch.
git commit -m "Description of the new feature"
```

(3) It may have taken you some substantial amount of time to develop the `newfeature` branch. During that time the GitHub `main` branch may have changed. So, prior to pushing your `newfeature` branch, merge the GitHub `main` branch into the local `main` branch, and then merge the local `main` branch into the local `newfeature` branch.

```
# Update the origin/* remote-tracking branches.
git fetch origin

# Make main the current branch.
git checkout main

# If necessary, merge the origin/main branch into the main branch.
# You haven't changed your main branch since the last push, so there
# should be no conflicts.
git merge origin/main

# Make newfeature the current branch.
git checkout newfeature
```

```
# If necessary, merge the main branch into the newfeature branch.
# There could be conflicts. Resolve them if necessary.
git merge main
```

(4) Push the `newfeature` branch, ask your teammates to comment on it, and eventually merge it.

```
# Push the newfeature branch to the origin repo.
git push --set-upstream origin newfeature
```

Browse to github.com.

Using the browser, initiate a GitHub "pull request" for the `newfeature` branch.

Using the browser, wait for comments from your teammate(s).

To assess your GitHub `newfeature` branch, your teammate(s) might:

```
git stash
git fetch origin
git checkout newfeature
Run tests.
git stash pop
```

If the GitHub `newfeature` branch is approved to your teammate(s), then (using the browser) accept the pull request, thus merging the GitHub `newfeature` branch into the GitHub `main` branch. If the approval process was reasonably quick, then there should be no conflicts. If there are conflicts, then go back to (3).

(5) Repeat Steps 1-4 for each new feature.

# Learning More

There is much more to Git. To learn more I recommend that you read the Git book at https://git-scm.com/book/en/v2, and the Git reference manual at https://git-scm.com/docs as required.

This document describes how to use Git through its command-line interface. An alternative is to use a graphical user interface. The page https://en.wikipedia.org/wiki/Comparison_of_Git_GUIs lists many Git graphical clients. Former COS 333 students have recommended *VS Code*, *GitHub desktop* (for Mac and MS Windows) and *TortoiseGit* (for MS Windows).

There also is much more to GitHub. I recommend that you investigate these features after you're comfortable with GitHub basics:

- **Projects**: for tracking goals and progress of a general "project". The official documentation is at https://help.github.com/en/github/managing-your-work-on-github/about-project-boards. A reasonable video tutorial is at https://youtu.be/ff5cBkPg-bQ.
- **Issues and Pull Requests**: for defining and implementing small program features in a collaborative and scalable way. An introductory guide is at https://guides.github.com/activities/hello-world/. A verbose set of documents about GitHub *flow* is at https://help.github.com/en/github/collaborating-with-issues-and-pull-requests.
- **GitHub Pages**: for website hosting directly from source code. GitHub's own Pages introduction is at https://pages.github.com/.

- **Actions**: for automatically running code like linters, builders, and deployers whenever you push to GitHub. The official documentation is at https://help.github.com/en/actions/automating-your-workflow-with-github-actions.
- **Tags and Releases**: for creating *versions* of your code both to release and reference. *Tagging* in Git is covered at https://git-scm.com/book/en/v2/Git-Basics-Tagging). Creating GitHub *releases* is covered at https://help.github.com/en/github/administering-a-repository/creating-releases.