Here, "remote" and "branch" have the same meanings as in the `git push` command. Good practice is to pull just before any time you push.

- *Origin and Main.* "Origin" and "main" are two terms you'll often see when working with Git.

  "Origin" refers to the default remote repository where your code is stored. When you clone a repository, Git sets up a remote called "origin" that points to the URL of the original repository. You can push/pull changes to/from "origin" to collaborate with others.

  "Main" is the name of the default branch in a Git repository. This is the branch where the main line of development occurs, and it's typically the branch you'll be working on most of the time. However, depending on the repository, the default branch may be named something else (e.g., "master").

Command	Explanation
`git status`	Display the staging area and untracked changes.
`git pull origin main`	Pull changes from the online repository.
`git push origin main`	Push changes to the online repository.
`git add <filename(s)>`	Add a file or files to the staging area.
`git add -u`	Add all modified, tracked files to the staging area.
`git commit -m "<message>"`	Save the changes in the staging area with a given message.
`git checkout -- <filename>`	Revert changes to an unstaged file since the last commit.
`git reset HEAD -- <filename>`	Remove a file from the staging area.
`git diff <filename>`	See the changes to an unstaged file since the last commit.
`git diff --cached <filename>`	See the changes to a staged file since the last commit.
`git config --local <option>`	Record your credentials (`user.name`, `user.email`, etc.).

Common Git commands.

Now that you've learned about some of Git's features, it's time to implement them in your first ACME Lab, *Intro to GitHub*!

2 Getting Started On Mac/Linux

Welcome to ACME! This guide will help you get set up for ACME.

Git

Git is a version control system that helps you manage changes to your code over time. This section will serve as a guide for the later steps, so do not worry about learning everything right now. It will come with time. Git allows you to keep track of different versions of your code, collaborate with others, and revert changes if necessary.

To install Git on WSL or another Linux system, run the following command in your terminal:

```
sudo apt install git
```

On Mac, you can just type `git` into your terminal and it will prompt you to install it. If you have an M1 Mac, you should also double check that you have OS version 12 or later.

Application in ACME

In ACME, Git is not only used to save and organize your work, but also to turn in assignments. The most current version of your code will be *pulled* by the instructor at a predesignated time and graded. If you always *push* your code when you're done working on it, you will never miss turning in an assignment. Explanations of these Git terms are provided later in this document.

Downloading Course Materials

The lab manuals can be found at `https://foundations-of-applied-mathematics.github.io/`. You can also download the zip folders that contain the lab materials from this page. You should then unzip these in a folder you are familiar with, such as your `Documents` or `Desktop` folder.

Achtung!

Make absolutely sure your unzipped lab folders are not nested! When you open each lab folder, you should NOT have to open another folder to access all the lab materials! If your folder is nested, move the inner folder out so that there's only "one layer" to access your lab materials.

Online Setup

1. *Sign up for GitHub.* If you already have a GitHub account, sign into your account and proceed to the next step. Otherwise, create a GitHub account at `https://github.com/`. This is where you will store your files online for grading.

2. *Create an ssh key.* This step only needs to be done once on each computer you want to use to access your repository. If you have multiple repositories on the same computer, you do *not* need to repeat this step for each one. To create an ssh key, enter the following command in the terminal:

   ```
   ssh-keygen -t ecdsa -b 256
   ```

   Press the `Enter` or `Return` key to accept the default file location. It will then prompt you to enter a Password, but you can press `Enter` or `Return` again to skip this step if you don't want a Password. The key will then be created. The file for the key will be placed in the `/home/<username>/.ssh` (or `~/.ssh`) directory.

   Now that the key is created, you need to add it to your GitHub account. From GitHub, click on your profile icon in the upper right corner, and click on **Settings** towards the bottom of the menu. Now click on **SSH and GPG keys** on the left, and click the green **New SSH key** button. Make a Title for this key (it doesn't matter what you put, but you may wish to specify which machine this key will be used for). Make sure the **Key type** is set to `Authentication Key`.

   Now, using the file explorer, navigate to the `.ssh` folder, and open the *public key* file. This file should be called `id_ecdsa.pub`; do *NOT* use `id_ecdsa` (without the `.pub` extension). Copy the contents of this file and paste it into the `Key` field on GitHub.

   If you're having trouble navigating to the `.ssh` folder, you can try to print the contents of the folder by running this code in your terminal:

   ```
   cat ~/.ssh/id_ecdsa.pub
   ```

   Once you've pasted the contents of `id_ecdsa.pub` into GitHub, press the green **Add SSH key** button. Then, in your terminal run

   ```
   ssh-add ~/.ssh/id_ecdsa
   ```

   If you get an error that says "Could not open a connection to your authentication agent," then run

   ```
   eval $(ssh-agent)
   ssh-add ~/.ssh/id_ecdsa
   ```

   To verify that this worked, when you make your first commit the new ssh key should be added to the file `~/.ssh/known_hosts`.

3. *Make a new repository.* You will need to create a new repository for each lab folder you downloaded. Note: you *must* follow these instructions exactly. If you do not complete each step exactly as specified below, delete the repository and start over.

   - On your GitHub Home page, click the green **Create repository** button to the upper left of the screen; or, if you have already used GitHub before and already have repositories set up, click on the green **New** button next to **Top Repositories**.
   - First you will need to provide a Repository name; this can be anything you want, but please make it relevant to the lab folder it will represent; you may also provide a repository description if you like.
   - Mark the repository as **Private**.
   - Leave the box next to **Add a README file** unchecked (if you accidentally include a `README`, delete the repository and start over).
   - Under **Add .gitignore**, select **None** for `.gitignore template` (if you accidentally include a `.gitignore`, delete the repository and start over).
   - Under **Choose a license**, select **None** for `License`.
   - Finally, click the green **Create repository** button.

   Repeat this process for each lab folder you downloaded.

4. *Give the instructor access to your repository (First Day of Class).* If you are doing this before the first day of class, skip to the next step. In your newly created repository, click the **Settings** tab along the top of the page and click on **Collaborators** on the left. Then click on the green **Add people** button. Type your instructor's GitHub username, select them, then click the big green **Add <username> to this repository** button.

5. *Connect your folder to the new repository.* In your terminal enter the following commands.

```
# Navigate to your folder.
$ cd /path/to/folder  # cd means 'change directory'.

# Make sure you are in the right place.
$ pwd                 # pwd means 'print working directory'.
/path/to/folder
$ ls *.md             # ls means 'list files'.
README.md             # This means README.md is in the working directory.

# Connect this folder to the online repository.
$ git init
$ git remote add origin git@github.com:<username>/<repo>.git
# Make sure the link has this form. If it starts with https, select the ↩
    ssh option on GitHub instead.

# Record your credentials. user.name should be your name, and user.email ↩
    must be the email associated with your GitHub account.
$ git config --global user.name "your name"
$ git config --global user.email "your email"

# Add the contents of this folder to Git and update the repository.
```

```
$ git add --all
$ git commit -m "initial commit"
$ git branch -M main    # Update branch name.
$ git push origin main
```

For example, if your GitHub username is `greek314`, the repository is called `acmev1`, and the folder is called `Student-Materials/` and is located in `Desktop`, you would enter the following commands:

```
# Navigate to the folder.
$ cd ~/Desktop/Student-Materials

# Make sure this is the right place.
$ pwd
/Users/Archimedes/Desktop/Student-Materials
$ ls *.md
README.md

$ git init
$ git remote add origin git@github.com:greek314/acmev1.git

$ git config --global user.name "archimedes"
$ git config --global user.email "greek314@example.com"

$ git add --all
$ git commit -m "initial commit"
$ git branch -M main
$ git push origin main
```

At this point you should be able to see the same files on your GitHub repository that are also found on your machine in your course directory. If you enter the repository URL incorrectly in the `git remote add origin` step, you can reset it with the following line:

```
$ git remote set-url origin git@github.com:<username>/<repo>.git
```

Note

You may get an error like the following when you run `git push`:

```
...
fatal: Authentication failed for 'https://github.com/<username>/<repo↩
    >.git/'
```

If this error occurs, your repository URL is in the wrong format; most likely, you used the **https** format instead of the **ssh** format shown above. You can use the `git remote set-url origin` command to fix this issue as well.

6. *Download data files.* Many labs have accompanying data files. To download these files, navigate into each of your course directories and run the `download_data.sh` bash script, which downloads the files and places them in the correct lab folders for you. You can also find individual data files through `Student-Materials/wiki/Lab-Index`.

```
# Navigate to your folder and run the script
$ cd /path/to/folder
$ bash download_data.sh
```

Repeat this process for each of your lab folders.

Installing Python and Dependencies

Before installing Python on Mac, you will need to install Homebrew (brew). To check if brew is already installed, run

```
which brew
```

If the answer is

```
/usr/local/bin/brew
```

then you can skip to the update / upgrade step, but if the answer is

```
brew not found
```

then you must first install brew by running the command found at brew.sh, which will probably look like

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/↩
    HEAD/install.sh)"
```

When brew finishes installing, it should tell you to run two commands to add Homebrew to your "path". This makes brew accessible to run from wherever you are in your file system. The commands should look something like

```
(echo; echo 'eval "$(/opt/homebrew/bin/brew shellenv)"') >> ~/.zprofile
(echo; echo 'eval "$(/opt/homebrew/bin/brew shellenv)"') >> ~/.zprofile
```

Run the two commands brew tells you to run. You can now update brew by running

```
brew update
brew upgrade
```

To install Python, navigate into one of your course directories (e.g. `Volume1`) and run

```
bash install_python.sh
```

Now restart your terminal.

The next step is to set up a "virtual environment." Using a separate virtual environment for separate Python projects makes it possible to isolate sets of dependencies from each other. This prevents packages for one project from inadvertently interfering with those of another project—for example, your ACME labs will need one set of packages, while another project you do in Python may require different versions of the same packages. Environments are considered good practice in industry, if not essential; for the small amount of extra effort, they can save a *lot* of headache.

We'll be using Python's native virtual environment system, `venv`, and we'll be setting up just one environment for all our ACME code (since there's just one set of dependencies). In the terminal, navigate to the folder where you'd like to keep your ACME lab folder(s), for example, `~/documents`. Then to create an environment in a folder named `.venv`, write the command

```
python -m venv .venv
```

To "activate" the environment—that is, to switch to the environment's version of Python together with its associated dependences—use the `source` command:

```
source .venv/bin/activate
```

Note

If you accidentally create the virtual environment folder in the wrong location or with a name you don't like, no worries! Just delete the folder and create a new one with the instructions above. **Do not** just move or rename the folder, as the environment contains scripts that will not get properly updated, and the environment will simply not work.

You should see your activated environment `(.venv)` appear at the beginning of the current line in the terminal, like `(.venv) <username>@<computer_name>:`. To deactivate the environment, simply write `deactivate`. You should see `(.venv)` disappear.

We're finally ready to install some Python packages to our environment. First, **make sure you've activated your new environment** and verify that `(.venv)` appears at the start of the current line of the terminal. Now run

```
bash install_dependencies.sh
```

The last line of the output should say "Python dependencies successfully installed!" If it doesn't, double-check that you've activated your virtual environment.

You have now installed Python and all necessary dependencies you will need in ACME!

Using VS Code

VS Code is the recommended code editor in ACME, though it is not required. We recommend it because it's free and open source. You can download it at `https://code.visualstudio.com/`. Once you have it installed, you can search for it on your machine under: `Visual Studio Code`. You can also open it from the terminal by typing `code`.

1. *Installing the Python Extension:* Search in the `Extensions` tab for “Python”, then click install. This will enable VS Code’s support for Python.

   It will also make it possible for VS Code to use your virtual environment. Press `Cmd+Shift+P`, then search “Python: Select Interpreter” and press `Enter`. If the path to your virtual environment doesn’t come up, click “Enter interpreter path...”, then “Find...”, then start by typing the path to your virtual environment folder. Select the Python file inside this folder, located at `<virtual_environment_folder>/bin/python`.

If All Else Fails. . .

If, while working on a lab, you can’t get something to work on your computer, your last resort is Google Colab. It should be reserved as a last resort, because Google Colab is formatted in a notebook style, which is fundamentally different than a `.py` file. However, if you are ever stuck, keep in mind that it’s still an option. See the Using Google Colab guide in the Appendix for more information on using Colab for ACME labs.

Additional Git Help

Here are some key concepts and terminology you’ll need to understand and use Git effectively. You may need to refer back to this multiple times.

- *Repositories.* A Git repository is a collection of files and folders that Git is tracking. By creating a repository, you are simply telling the software that it should back up certain files that you tell it to. When you create a repository, Git creates a hidden directory called `.git` inside your project folder that tells it to track those files.

- *Adding Files.* Git will only track the files that you specifically add. You can add files individually, or you can add all the files in a folder at once. To add a file, you’ll typically use the command:

  ```
  git add <filename>
  ```

- *Commits.* A commit is a snapshot of your code at a specific point in time. When you make changes to your code, you can create a new commit to record those changes. Each commit has a unique identifier, which allows you to reference it later if you need to revert your code to a previous state. If you are collaborating with others, it gives you an “undo” button specific to each user, allowing you to be more organized. It is good practice to commit every time you leave your computer to keep track of your work.

  To create a commit, you’ll typically use the command:

  ```
  git commit -m "Commit message"
  ```

- *Branches.* A branch is a separate line of development that diverges from the main line of development. By creating a new branch, you can work on a feature or bug fix without affecting the main codebase. Once you're done with your changes, you can merge the branch back into the main codebase.

You probably won't branch your code off the main branch in the ACME courses, but this term is still useful to know.

- *Cloning.* Cloning a repository is simply downloading a codebase. Just like downloading a zip folder, all the files are present, but the original author does not have access to your files unless you push them back.

  To clone a repository, you'll typically use the command:

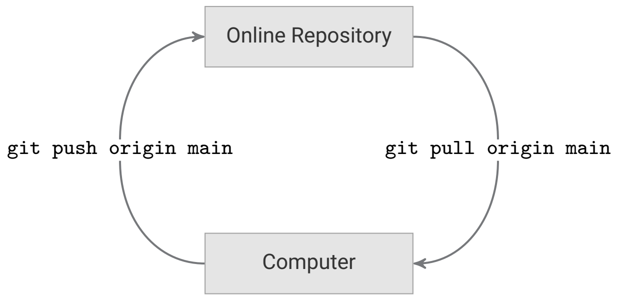
```
git clone <url>
```

- *Pushing and Pulling.* Among the most frequent operations you'll use with Git are pushing and pulling.

  When you *push* changes to a remote repository, you're sending your commits to a central server where others can access them. This will be where the grader accesses your code in order to give you a grade. To push your changes, you'll typically use the command:

```
git push <remote> <branch>
```

Here, "remote" refers to the remote repository where you want to push your changes (e.g., "origin"), and "branch" refers to the name of the branch you're pushing (e.g., "main"). Good practice is to push every few hours of working, and absolutely every time you're done with a lab. Pushing saves your work on a secure server, meaning if something catastrophic happens to your computer, none of your work will be lost and you can continue where you left off on a different machine. Do not assume you're different and that computer issues will never happen to you. Doing a lab from scratch twice in a week is guaranteed to take more time than just once!



Exchanging Git commits between the repository and a local clone.

When you *pull* changes from a remote repository, you're downloading changes that others have made and incorporating them into your local codebase. This will be used to pull your grading feedback back from the grader. To pull changes, you'll typically use the command:

```
git pull <remote> <branch>
```

Here, “remote” and “branch” have the same meanings as in the `git push` command. Good practice is to pull just before any time you push.

- *Origin and Main.* “Origin” and “main” are two terms you’ll often see when working with Git.

  “Origin” refers to the default remote repository where your code is stored. When you clone a repository, Git sets up a remote called “origin” that points to the URL of the original repository. You can push/pull changes to/from “origin” to collaborate with others.

  “Main” is the name of the default branch in a Git repository. This is the branch where the main line of development occurs, and it’s typically the branch you’ll be working on most of the time. However, depending on the repository, the default branch may be named something else (e.g., “master”).

Command	Explanation
`git status`	Display the staging area and untracked changes.
`git pull origin main`	Pull changes from the online repository.
`git push origin main`	Push changes to the online repository.
`git add <filename(s)>`	Add a file or files to the staging area.
`git add -u`	Add all modified, tracked files to the staging area.
`git commit -m "<message>"`	Save the changes in the staging area with a given message.
`git checkout -- <filename>`	Revert changes to an unstaged file since the last commit.
`git reset HEAD -- <filename>`	Remove a file from the staging area.
`git diff <filename>`	See the changes to an unstaged file since the last commit.
`git diff --cached <filename>`	See the changes to a staged file since the last commit.
`git config --local <option>`	Record your credentials (`user.name`, `user.email`, etc.).

Common Git commands.

Now that you’ve learned about some of Git’s features, it’s time to implement them in your first ACME Lab, *Intro to GitHub*!

A Using Google Colab

Lab Objective: *Google Colab is an environment similar to Jupyter Notebooks that is run remotely on Google's servers. It has some advantages over other environments, including easy package management and installation and the ability to run on a machine that doesn't have python installed. This appendix details how Colab can be used for ACME labs, as well as some issues to be aware of and relevant workarounds.*

Setup

Google Colab can be started by going to the Colab website `https://colab.research.google.com/` or by opening a `.ipynb` file from your Google Drive. When Colab starts, you can upload a `.ipynb` file by selecting the `Upload` tab and choosing the `.ipynb` file you wish to upload. Colab can also be used with `.py` files, but the process is somewhat more involved and is detailed below.

Once your file is open in Colab, it can be run and edited like a normal Jupyter notebook. To submit your lab for grading, you will need to download the file by selecting `File > Download` and then either `Download .ipynb` or `Download .py`. You will then need to move your file into the repository clone on your machine in the correct lab folder, where you can then add, commit, and push it like normal.

Achtung!

Remember to keep the name of the file ***IDENTICAL*** to the original name of the lab spec! Failure to do this will cause the test driver to skip over your file and give you an automatic zero for the lab!

Using .py Files with Colab

Google Colab only works with `.ipynb` files, but many ACME labs use `.py` files. Since the former filetype has additional formatting, there isn't a way to directly import a `.py` file into Colab. The simplest way around this is to create a new file in Colab and copy-paste the lab file's contents into it. This way, it can be edited and used as a normal Jupyter notebook. Then, when you're finished, download the notebook as a `.py` file instead of a `.ipynb` file.

However, under these circumstances some care must be taken in order for your work to be graded properly. In particular, before you download your file, make sure that the following are satisfied:

- All Colab-specific code to upload files is commented out; as the packages used do not exist outside of Google Colab, the test driver will raise an error when it attempts to import them.
- There are no calls to the lab's functions, for testing purposes or otherwise, outside of a typical `if __name__=="__main__"` statement; otherwise, they will be run when the file is imported by the test driver, which will certainly cause issues.

Using Data Files

To use a data file in Colab, you must first upload it by running the following in a code cell in Colab:

```
from google.colab import files
files.upload()
```

When run, this will create a prompt for you to upload your files. After doing so, the data files can be accessed from the notebook as normal.

The best way to upload an entire folder is to first zip it, then upload the zipped folder to Colab, and then unzip it in the Colab notebook:

```
# To upload a folder called 'data', first zip it, and then upload 'data.zip'
files.upload()
# Then, it can be unzipped
!unzip data.zip
```

If the files do not need to be in the same folder, you can always simply upload multiple files at once with the usual `files.upload()` code.

Installing Packages

One advantage of Google Colab is that a very large percentage of the python packages used in the ACME labs are already pre-installed. However, there are a few packages that are not installed, and must be installed each session. Packages can be installed by running `pip` in a code cell, which is done by putting an exclamation point before the usual command. For example, to install the GeoPandas package, you would run:

```
!pip install geopandas
```