# Bash and Git Workshop Tutorial

Luke Colosi
Software Carpentry Workshop at Scripps Institution of Oceanography

August 29, 2022


Figure 1: Bash Shell.

## Contents

# 1 The Unix Shell: Summary of the Fundamental

The Bash shell gives you power to do simple jobs more efficiently and in a timely manner in order to streamline researcher's day to day tasks. These jobs can range from moving files, directories, and data locally on your computer to communicating with remote servers and many other data management tasks. Bash, like many other coding languages, has a large repository of built in and downloadable commands (via Homebrew for MacOS) which can be ran in the command-line, in screens, and through shell scripts. For general reference to any of the Bash topics covered in this tutorial, use the software carpentry workshop website.

## 1.1 Introducing the Shell

For MacOS, the Bash Shell is ran through the terminal which is located by pressing command + space bar (spotlight search) and typing terminal into the search bar. Once you select the terminal, the terminal interface should pop up and you will see two lines (Figure 2):

1. **Line 1**: Last login date and time
2. **Line 2**: (from left to right) What computer you are on (indicated by Johns-MacBook-Pro-7), what directory you are currently in (indicated by the tilde) and what user you are currently login under (indicated by lukecolosi in Figure 2).

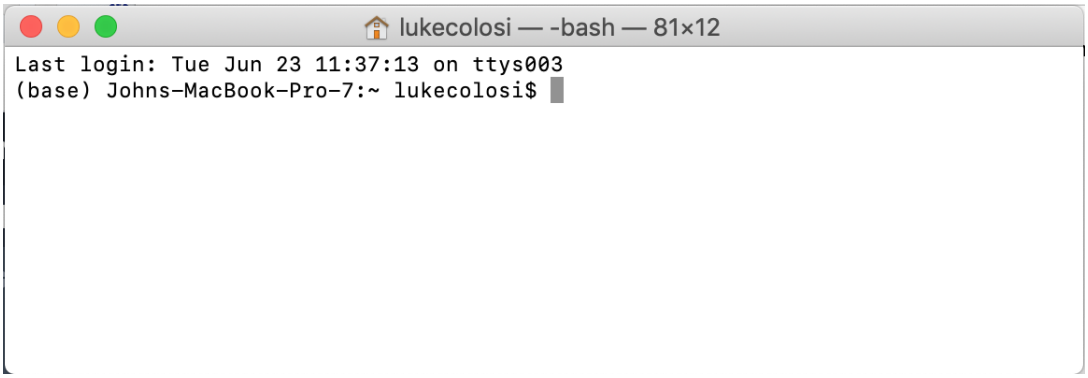


Figure 2: Bash Shell Interface when first booting up the terminal

All of the subsequent information in this tutorial will begin from this point in the terminal or bash shell. However, before beginning your work in the bash shell, it is import to download all necessary commands that are not automatically available on your local computer desktop. For MacOS, install Homebrew to obtain missing packages. In addition, it is important to install XCode and XQuartz for enabling graphical interface.

Here are a few tips for making your learning experience in the terminal as easy as possible.

1. You cannot use the curse to change the place where you typing. You must scroll over to another location in the command line.

2. To simplify the computer, directory, and user information on the left hand side of the $ symbol, type into the command line: PS1='$'
3. The gray box indicating where your are typing is not the place where you are actually typing. You are typing the space before for this gray box.
4. There are many built-in short cuts for the bash shell. Here is link for an extensive list.
5. The up-arrow key is used to scroll up through previous commands to edit or repeat them. In order to search the history of previously entered commands use Ctrl + R. In addition, history command displays recent commands and typing number repeats a command by the number order it was preformed.
6. From now on, **never name a file with a space in it!** If you try write a path with a directory that has a space in the title, you will need to put a backward slash followed by a space to not get an error. For example, to change directories to the directory To do lists, we would have to write cd To\ do\ lists. We would get an error from these spaces because the command cd would consider each word in the directory's name as a separate argument. Instead of a space, use the underscore symbol.
7. To look at documentation for any command in bash use either man command or command – – help.
8. **MOST IMPORTANT TIP**, the tab key will auto complete a partially written command or path. Use this a much as possible to make your life easier.

## 1.2 Navigating Files and Directories

In order to manage all the information on your computer (on your disk) the information is organized into a file system. Information is stored in files while files are stored in directories. The file system can be though of as a tree with many branches (directories) extending to smaller branches (directories within directories) and then eventually to leaves (files) which makes up the file system hierarchy (Fig. 3).

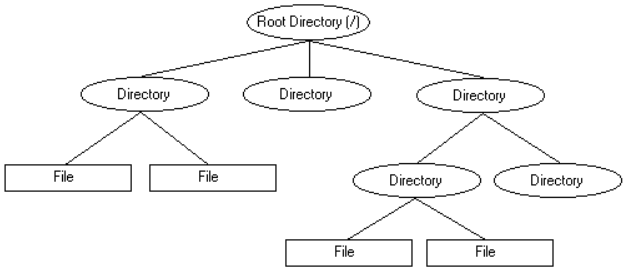


Figure 3: File system Illustration.

At the top of the file system (the trunk) is the root indicated by the first fore-ward slash "/". Branching off from the root are directories and possibly files that serve many purposes. For a

standard Unix file sytems, these directories include essential user command binaries (/bin), configuration files for the system (/etc), essential system binaries (/sbin), read-only user application support data and binaries (/usr), variable data files (/var), user home directories (/home) and many more. Fig. 4 summaries the standard Unix filesystem.

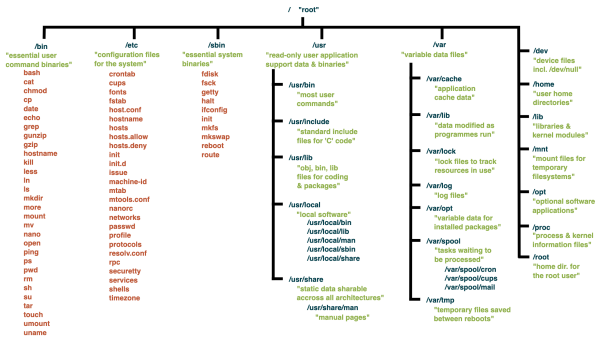


Figure 4: Standard Unix filesystem structure and descriptions.

Within these directories are other directories and file. When naming files and directories, never use spaces between words in the title. Instead, use the underscore for spaces. Also avoid quotes or wildcard in filenames and make sure to give files consistent names that are easy to match with wildcard patterns to make it easy to select files when looping. In addition, avoid using capitalization because it is just one more key you will have to press when typing out the file name.

In order to navigate the file system, three basic commands are used: cd (change directory), ls (list), and pwd (present working directory) as displayed in the table below.

| Navigational Bash Commands | |
|---|---|
| cd path | changes the current working directory |
| cd .. | move up in the file system hierarchy by one directory towards the root |
| cd ~ or cd | move to your home directory |
| ls path | prints a listing of specific files or directories for the given path |
| ls | lists the current working directory's files and directories |
| pwd | prints the user's current working directory |

The path referenced in the table above is the route through the file system which the computer takes in order to preform a command. If the path begins with the root, the path is

call an absolute path. If the path starts from the current directory (excluding all directories of higher file system hierarchy), the path is call a relative path. Between each directory in the path is a forward slash.

Note that all these commands can be slightly changed in order preform specific tasks. These tasks are specified with a flag which a commonly represented with a hyphen and a letter or letters. For example, the command ls has the flag -la which tells the bash shell to output a list of all directories that are visible or hidden (directories that begin with a period are not visible with the usual ls command) and the flag -F which tells the bash shell to output a list of all contents of the current directory and specifies the type of each (i.e. backslash for directory, asterisk for executable scripts, etc.).

## 1.3 Working with Files and Directories

When working with files and directories, there are other list of key commands you must become familiar with. These are displayed in the table below.

| File and Directory Manipulation Commands | |
|---|---|
| cp old_path new_path | copies a file from a old path to a new path |
| mkdir path/name of directory | creates a new directory at the location specified by the path and with the name given at the end of the path |
| mv old_path new_path | move a file or directory from an old location (old path) to a new location (new path) |
| rm path | removes (deletes) a file |
| rm -r path | removes (deletes) a directory |
| cat path_to_file | displays all content within a file |

If the old path is the same as the new path for the mv command, then the file can be renamed by writing the old name of the file at the end of the old path and the new name of the file at the end of the new path. Path can be generalized to include many files or directories that have a common phase, word, or letter/number by using wildcards. The asterisk wildcard (*) is used to match multiple characters in a filename with other filenames. For example, *.txt matches all file names that end with .txt. The question mark wildcard (?) matches any single character in a filename with other filenames. For example, ?.txt matches with a.txt but not with any.txt.

Important features to note about the bash shell:

1. The shell does not have a trash bin, so once a file or directory is deleted, the information is gone. So be extremely careful with the rm command!
2. Filenames come in the form name.extension. The extension indicates the type of data in the file. For example, an image can have the extensions .jpeg or .png while oceanographic data files can have the extensions .txt (text file) or .nc (netcdf file).

## 1.4 Pipes and Filters

Pipes join multiple commands within one command line entry. Basic syntax for a combining two commands with a pipe:

Command1 | Command2 ⇒ run command 1, take output from command one and run command 2 with output

For example, the command:

wc -l *.txt | sort -n | tail -n 1

outputs the number of lines in multiple files ending with .pdb, sorts the output numerically, and displays the last line. Here are a few other useful commands:

head ⇒ display first lines (first 10 as default) of a file.
tail ⇒ display the last part (last 10 lines as default) of a file.
sort ⇒ sorts text and binary files by lines.
wc ⇒ counts lines, words, and characters in its inputs

These commands filter specific information present in the file. To save output into a new text file, use:

Command >> filename.txt

To redirect a command's output to a file and overwrite any existing content, use:

Command > filename.txt

## 1.5 Loops

The Loops focused in this section are "for" loops. A for loop iterates with a loop variable through a list of objects. For each iteration, the loop variable is assigned in order a new object in the list. The general structure of a for loop is illustrated in Fig. 5.

The symbol $ must come before the loop variable in order for bash to recognize it as a variable. The list can consist of numbers, words, or both. If the list is not too long, you can type out the list by hand. For example, Fig. 6 illustrates a simple for loop.

```
for thing in list_of_things
do
    operation_using $thing
done
```

Figure 5: For loop structure in the Bash Shell

```
$for number in 01 02 03 04 05
> do
> echo 'filname_'$number'.txt'
> done
filname_01.txt
filname_02.txt
filname_03.txt
filname_04.txt
filname_05.txt
$
```

Figure 6: Simple for loop example.

If the list is too long to type out by hand, use the command for number in 0..10 where 0 is the starting number and 10 in the final number in the list. Fig. 7 an example of a for loop calling a file and copying it to the local computer in.

```
#set variables
ftp=ftp://ftp.ifremer.fr/ifremer/ww3/HINDCAST/GLOBAL

#create a loop that will go through each file in Ifremer to obtain hs for 2004-2011 for the
#European ww3 model run.
for y in `printf "%02d " {12..16}`
do
     d=20${y}_ECMWF
    `wget -c -r -np -R "index.html*" -nH --cut-dirs=7  ${ftp}/${d}/hs`
done
```

Figure 7: Complex For loop example.

## 1.6 Text editors and Shell Scripts

There are two types of text editors we will go over, vim and nano. Note that there are many more with a broad range of capabilities and syntax. For both vim and nano, you can initiate a new file with any specified extension by the following or access a preexisting file:

vim filename.extension
nano filename.extension

Beginning with vim, there are two modes: insert and command modes. To enter the insert mode used for writing text in the document, you can use one of the following commands:

1. **a**: Append text following current cursor position.
2. **A**: Append text to the end of current line.
3. **i**: Insert text before the current cursor position.
4. **I**: Insert text at the beginning of the cursor line.
5. **o**: Open up a new line following the current line and add text there.
6. **O**: Open up a new line in front of the current line and add text there

To enter the command mode press the escape key (esc). The following commands when in command mode are cursor movement, exit, text deletion, and Past commands. The most important of these are listed below:

**Cursor Movement Commands**

1. **h**: Moves the cursor one character to the left
2. **l**: Moves the cursor one character to the right
3. **k**: Moves the cursor up one line
4. **j**: Moves the cursor down one line
5. **$**: Move cursor to the end of current line
6. **0**: Move cursor to the beginning of current line

**Exit Commands**

1. **:wq**: Write file to disk and quit the editor
2. **q**: Quit (a warning is printed if a modified file has not been saved)
3. **ZZ**: Save workspace and quit the editor (same as :wq)

**Text Deletion Commands**

1. **:dn**: Deletes n number of lines after the cursor. n is any positive integer number.
2. **d$**: Delete to end of line.

**Paste Commands**

1. **:set paste + i**: Used to enter insert paste mode.
2. **u**: Undo last change.
3. **U**: Restore line.

For a full list of vim editor commands, refer to the user command guide *here*.

For nano, I will no go into details. Rather, for a full list of nano editor commands and description, refer to the user command guide *here*.

Throughout a research project, scientists will need to write blocks of code in bash for a large assortment of jobs including download large data with a for loop or extract specific data dimensions. The best way to save and run these blocks of code is to create a shell script. A shell script are analogous to python scripts but for data management instead of data analysis. To initialize a shell script, use a text editor:

vim filename.sh

with the file having the .sh extension. To execute the shell script, we first need to make the script executable. This is because by initializing the script with the vim editor and adding the extension .sh, we are only suggesting to the computer that the file is a shell script. We will need to formally make it a shell script using the following:

chmod +x filename.sh

Now we can execute the script with the following command:

bash filename.sh or ./filename.sh

Note that bash filename.sh runs whether the file is executable or not and ./filename.sh only runs when the file is executable.

## 1.7 Finding Things

A great way to find the absolute path to files or directories in bash is using the find command. The find command has the following syntax:

find int_directory_path -name 'name' -type f or d

where

1. **int_directory_path**: The initial directory which your search will start from. The search will be preformed on sub-directories under the initial directory. For example, int_directory_path = . begins the search in the current directory, int_directory_path = / begins the search at the root of the computer so all directories are searched, or int_directory_path = begins the search at your home directory.
2. **'name'**: Name of the file. Quotes only need to be used if the title of the file or directory has spaces in it. Wild cards can also be used. For example, name = '*.txt' searches for all files with the extension .txt.
3. **f or d**: The -type flag specifies if the object being earched for is a file or directory. f is used if the object is plain text file and d is used if the object is a directory.

For more information, look use references 1 and 2 here.

Also, if you need to search for past commands you have done use the short cut crtl + R or simply scroll up with the up arrow.

## 1.8 Structuring a Project's directory

When beginning a project, a home directory for your project need to be made so all your documentation, code, references, figures, and data for that project is located in one place. Therefore, create a project directory and the following sub-directories:

mkdir project_name cd project_name mkdir src data docs figs notebooks

where the sub-directories are the following:

1. **src**: Source Code directory. This includes python, matlab, and bash scripts.
2. **data**: Data for project. Make sure not to download to much data onto your local computer. Use either a remote computer to store large amounts of data or external storage.
3. **docs**: Documents and references for your project. These can include research paper reference articles, research paper, presentations, etc.
4. **figs**: Figures for the project.
5. **notebooks**: Python notebooks for presenting to advisor.

For documentation purposes for the project. A README.md file should be made to document the outline of the project, funding, directories within project directory and their structure, and any important information that you think you may forget about that is critical for navigating through this project's directories.

## 1.9 Using awk

I will not go into detail about awk. Rather, here is a good reference user guide to awk.

## 1.10 Managing Environments with conda

For reference: link.

# 2 Version Control with Git

For a general reference for all Git related topics, use this link. An additional cheat sheet to Git commands can be found here. Here are a few precautionary warnings before using Git:

1. GitHub is used for two main purposes:
   (a) Version Control or in other words, to track changes on mutable documents.
   (b) Collaboration with peers and colleagues on projects.

   This means that that documents and data that are public and easily accessible remotely on websites or servers should not be placed on GitHub. In addition, GitHub gives a user limited storage. Therefore, it is vital that space on the repository is used efficiently. Disclaimer: This does not mean that you should never place data or immutable documents on GitHub. There are cases where these documents are not easily accessible and would be therefore beneficial to other GitHub users.
2. Make sure the documents, figures, data, and code that you place on a repository are your own work. If they are not, you must give credit to the author as well as obtain permission from them to have their work public on GitHub. Alternatively, you can make your repository private.

To install git on your computer follow the git installation guide in the link here.

## 2.1 Automated Version Control

Version control with Git allows programmers to collaborate and work in parallel on the same code, text files, small data sets, and figures while tracking changes made and storing history for past steps of the project. Here are three advantages for using version control with Git:

1. Nothing that is committed to version control is ever lost, unless you work really, really hard to delete it and past history. Since all old versions of files are saved, it is always possible to go back in your history to see exactly who wrote what on a particular day, or what version of a program was used to generate a particular set of results.
2. Because we have this record of who made what changes when, we know who to ask if we have questions later on, and, if needed, revert to a previous version, much like the "undo" feature in an editor.
3. When several people collaborate in the same project, it's possible to accidentally overlook or overwrite someone's changes. The version control system automatically notifies users whenever there's a conflict between one person's work and another's.

## 2.2 Setting Up Git

When you use Git for the first time on your computer, you will need to configure some of the setting. Use **git config** command with the flag –**global** to configure your settings and preferences for every project you do in your Git user account for your computer. Do this only **once per machine**.

Here are some of the necessary configuration commands you need in order to properly set up Git on your computer:

| **Git Set-Up Commands** | |
|---|---|
| **git config --global option** | Configures git for first time users globally (for all projects) with a operation option. |
| **git config --global user.name "Luke Colosi"** | Sets user name for subsequent Git activity. |
| **git config --global user.email "lcolosi@ucsd.edu"** | Sets email for subsequent Git activity. |
| **git config --global core.autocrlf input** | Changes the way Git recognizes and encodes line endings (for Mac OS). |
| **git config --global core.editor "vim"** | Sets default text editor with Git activities. |
| **git config --list** | Accesses a list of your current settings. |
| **git config -h** | Accesses the list of Git commands. |
| **git config --help** | Accesses the Git manual. |

## 2.3 Creating a Repository

Now that the Git settings are properly configured, we need to create the project's directory in an appropriate location on your computer using the command **mkdir project_name**. To tell Git to make the project directory a repository (i.e. a place where Git can store versions of your files), move into the project directory and use the command:

git init

To check the status of the repository (i.e. what files are being tracked and what modifications have been made, use the command:

git status

Nested repositories should be avoided at all cost because the repository higher up in the file system hierarchy tracks changes for all files in the current director, sub directories, etc. making the nested repository unnecessary. In addition, the two repositories could interfere with each other.

If a nested repository is made, one can delete this repository with the following steps:

1. Removing files from a git repository using **rm -f filename**.
2. Removing directories from a git repository using **rm -rf directory**.
3. Remove the .git folder using **rm -rf directry/.git**

To look at where the repository is located, use the list command ls -a in order to print hidden files such as .git.

## 2.4 Tracking Changes

Following the best practices guide for project structure discussed in the section 1.8, make sub-directories and README file within the project directory. When creating the README file, we can use either the .txt (text), .md (markdown), .yml (YAML) or .html (HTML) extension. Each type of file has different syntax for formatting and manipulating the file in order to get aesthetically pleasing outputs on GitHub. For now, we will use markdown files for the README file. For a reference to the Markdown syntax, use the link. For a reference to HTML syntax, use the link. For a reference to YAML syntax, use the link. To create the readme file, use any text editor your would like. This does not need to be the same at your default editor specified in the global configurations of your system. For now, I will use vim:

vim README.md

In the readme file, document the name of repository, date and location established, and general purpose and structure of the repository. An easy way of looking at the content of the readme file in the command line is to use **cat readme.txt**. By using the git command:

git status

the command line print a lists of all new or modified files to be committed and files that are tracked and untracked. Tracked means that changes to the file are being documented and saved. At the beginning, all file are labeled as untracked. In order to track a file and to take a snapshot of a file in preparation for versioning, use:

git add filename

We can also stage all new, modified or deleted files using the command **git add -A**. However, be careful to not accidentally stage an unwanted modification. To record staged changes made to files in the repository (staged changes are also called changes that have been "added to the glass") and commit the file snapshot permanently to the version history in the .git directory, use:

git commit **or** git commit -m "Description of Changes"

The first command opens up the readme.txt text file with a text editor and within the text editor, the changes can be documented (this options allow one to give a longer statement about their modifications). The latter is a short cut that includes the short specific description of changes in the command line text without going into the text editor. To make sure everything is up to date and the changes have been committed, use **git status**. Make sure that you are committing changes frequently. Frequently being every time you are able to say "I can check something off my to do list". This could be writing the introduction to your research paper or completing a Jupyter Notebook for data you are analysing. Do not commit too frequently such that you are saving unnecessary snapshots of your documents, but no too infrequently such that you lose versions of your documents that you may have wanted to go back to. To look at the project history to see what we have done, use:

git log

**git log** give a list of all commits made to the repository in reverse chronological order including the commit id number, author, time it was committed, and the log message. Use the command **git log -N** where N is an integer number spacifying the number of commits you would like to see. If more changes are made to files, then the changes need to be staged "on the glass" with the command **git add filename** and then committed to the version history with **git commit -m "Description"**. This process of staging and then committing is illustrated in Fig 8. In order to look at the exact changes made to all files or a single file (i.e. the difference between the current version and the last saved version) and to review them before staging and committing, use:

git diff **or** git diff filename.ext **or** git diff HEAD∼N filename.ext

The first command outputs all changes made to all modified files. The second outputs all changes made to a specified file. The third command looks at the difference between the current version and file version N commits before. For example, git diff HEAD∼1 README.md displays the changes made to the README.md file since one commit before the last saved version. In the output message, the first three lines tell you what file is being considered. Focus on the fourth line and on which tells you what has been changed in the file. To look at the changes of stages files, use **git diff –stage**.

In summary, Fig 8 shows that beauty of Git version control system. Git system control can be thought of as a snapshot machine. A file (text file, jupyter notebook .py file, etc.) can be created, and changes can be made. Any changes that you would want to record in the version history under .git will be placed in a staging area (git add) where the snapshot of these changes will be made. Once the snapshot is made (git commit), then the version of the file is saved and a description of the changes made to the document are documented on the README file. The staging area allows one to specify specific changes to be recorded to the version history instead of recording all changes that have been made to multiple files.

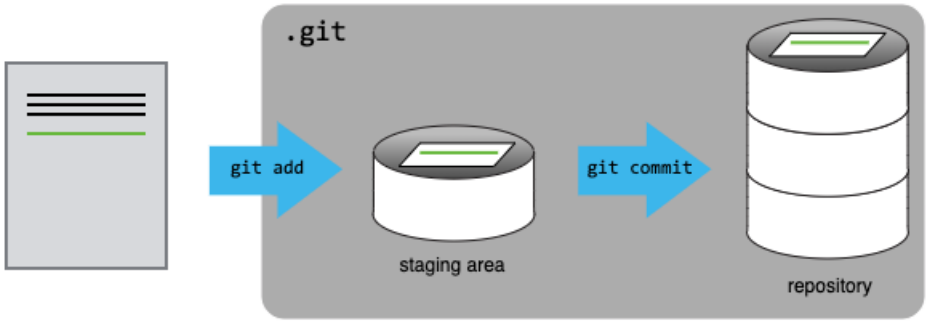


Figure 8: Structure and Mechanics of Git Version History Saving Process

## 2.5 Exploring History

Another way to look at the difference between a specific commit and the working directory is to use the commit ID which can be accessed using the command **git log** in the **git diff** command:

git diff f22b25e3233b4645dabd0d81e651fe074bd8e73b filename.ext

Or we can simplify the commit ID number to the first seven digits: **git diff f22b25e filename.ext**. To look at what changes we made at an older commit as well as the commit message, use the command:

git show

Now, suppose we have change our mind about the last change we committed and we want to restore the file back to the last saved commit. To restore the file back to the last saved version, we use the command:

git checkout HEAD filename.ext

Now, say we want to restore the file back even further to an older commit. In this case, we can either use **HEAD~N** or the truncated commit ID number:

git checkout f22b25e filename.ext

In summary, **git checkout** checks out (i.e. restores) an old version of a file. To illustrate this pictorially, Fig. 9 shows the process of retrieving the second to last version of FILE1.txt and FILE2.txt from the .git directory or branch master using the git checkout command. To show a list of the currently tracked files in the git version history, use:

git ls-tree -r master –name-only

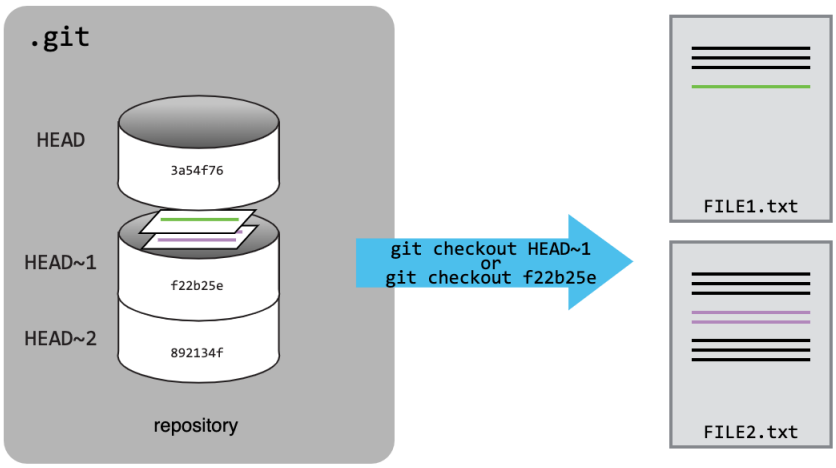


Figure 9: Illustration of retrieving the second to last version of FILE1.txt and FILE2.txt from the .git directory.

## 2.6 Ignoring Things

Suppose we have created or moved files into our project's root directory or sub-directories which we do not want git to track because it will be a waste of space or it distracts us from other important changes we have made. To ignore these files, we first need to create a file in the root directory of project named **.gitignore**:

vim .gitignore

In side the .gitignore file, we simply put the names of the files that we would like git to not track (on their own line in the file). Additionally, we can ignore entire directories by writing **directory_name/** or use wildcards for ignore multiple files with a particular filename or extension. For example, ***.png** will ignore all files with the extension .png.

Once .gitignore is created and modified, we need to add and commit .gitignore to the .git respository:

git add .gitignore
git commit -m "Ignore data files and the results folder."

Additional commands include the following:

1. **git add -f filename.ext** : Forces git to stage a ignored file.
2. **git status –ignored** : Outputs the status of ignored files or directories.

To remove files from git's version history, use the following:

1. Remove from local and remote repositories:

git rm 'file name'
git commit -m 'message'
git push

2. Remove from remote repository only:

git rm –cached 'file name'
git commit -m 'message'
git push

For more information, look here or here. This same process can be used when tracked files have been deleted and their history needs to be erased from the version history.

## 2.7 Remotes in GitHub

Systems like Git allow us to move work between any two repositories in order for multiple people to access the file in parallel and make changes. In practice, though, it's easiest to use one copy as a central hub, and to keep it on the web rather than locally on someone's laptop. The host server where the master copy of files for the repository are held is usually GitHub or Bitbucket. We will focus on GitHub.

To create this master copy of your local repository on GitHub, login into your github account and press the top left green new button as indicated by the red box in Fig. 10 or the pulse in the top right as indicated by the green box in Fig. 10.

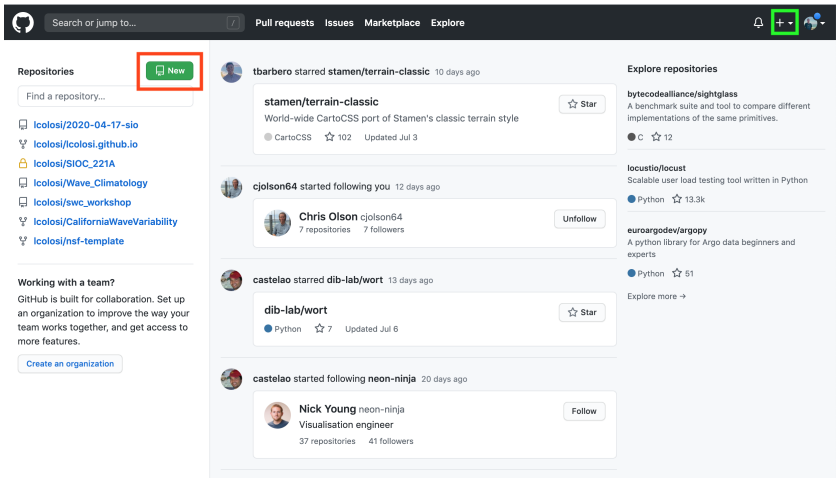


Figure 10: Creating a remote Git repository on GitHub.

This initiates the creation of a new repository that will show up under your github account. Next, name the repository (the same name as the local repository), give a description of the repository, and decide whether the repository with be public or private. Make it public unless you and uploading sensitive information such as solutions to homework problems for a class. Since this repository will be connected to a local repository, it needs to be empty.

Leave “Initialize this repository with a README” unchecked, and keep “None” as options for both “Add .gitignore” and “Add a license”.

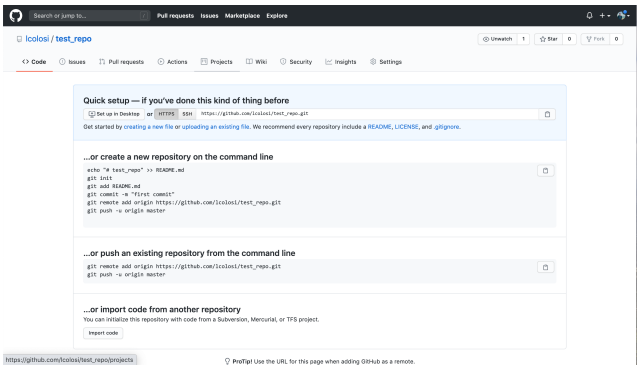


Figure 11: Instructions on how to connect local and remote servers.

Next a page (Fig. 11) will show up with three options of how to proceed locally on your computer in order to connect the remote and local repositories:

1. Create a new repository in the **command line**.
2. **Push an existing repository** from the command line.
3. **Import code** from another repository.

Because we have already created a repository locally, we will choose the second option. In addition, you can choose between HTTPS and SSH protocols. We will focus on the HTTPS protocol and connecting a local and remote repository. Now, we have two repositories: the **local repository** on your computer containing our files and the **remote repository** on GitHub which is currently empty. Our goal here is to link these two repositories together such that other people will have access to your work. Fig 12 illustrates these two repositories before they are connected.

Next, copy the URL from the browser in Fig. 11, go locally to the directory which the Git repository is located and run the command:

git remote add origin https://github.com/lcolosi/repo_name.git

This connects the local repository with the remote repository. Origin is a local name used to refer to the remote repository. To check if the command worked, use:

git remote -v

which will output the remote repository's URL. Once the remote is set up, this command will push all tracked files from our local repository to the remote repository on GitHub:

git push origin master

This is illustrated in Fig. 13. Once we make future changes to our files or add new directories and files to our local repository, we will need to go through the following process:

1. Staging changes, untracked files or deleted files: **git add -A**

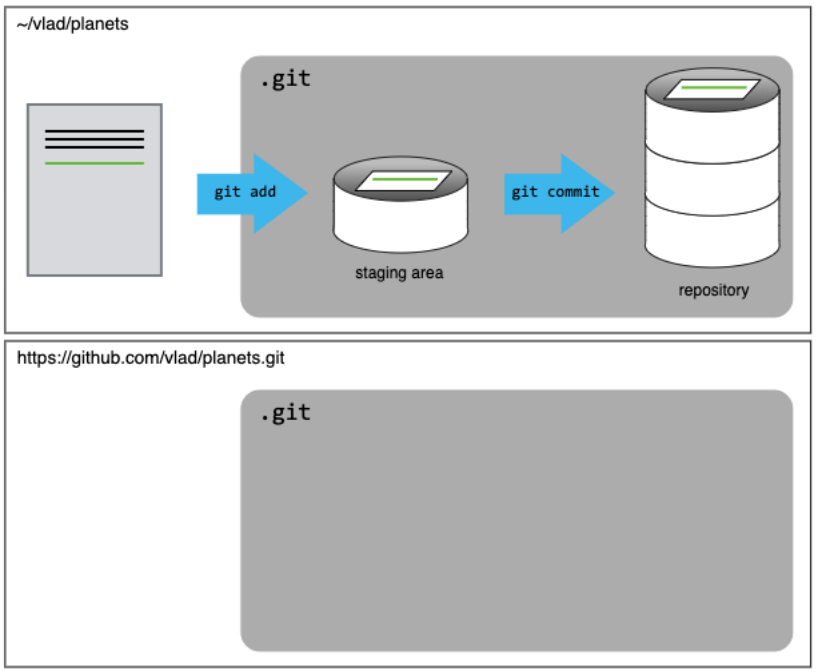


Figure 12: Connecting a local repository with a remote repository on Git Hub: The before connecting repositories stage.

2. Commit changes to the local Git repository: **git commit -m "Brief message."**
3. Push local changes to the Git repository to the remote repository on GitHub: **git push origin master**

We can also pull changes from the remote repository to the local one using the command:

git pull origin master

**THIS IS CRITICALLY IMPORTANT!** We should always do a git pull before we start making any changes to files on a repository. If we do not and there were changes made to files on the remote repository, then when we go to push our own changes to the remote repository there will be a conflict. See the Conflicts section for help to remedy this problem if you ever find yourself in this unfortunate situation.

Lastly, github has made some alterations to the username and password authentication (see link here for details). Originally when you **git clone**, **git fetch**, **git pull**, or **git push** to a remote repository using HTTPS URLs on the command line (like when connecting your local repository to a remote repository), Git will ask you for your GitHub username and password. However, since August 2021 the mode of authentication has changed to more secure authentication method than password-based ones. These include getting a personal access token (PAT) or using a the git credential manager. For more information on how to create and use a PAT, see the link here. For more information on how to install and use the git credential manager, see the link here. The downloading process requires using

homebrew, so make such you that installed and updated. Once homebrew is installed, a window will pop up whenever you're credential are required for a git command you try to execute in the command line. This window will ask you to sign into you github account for authentication.

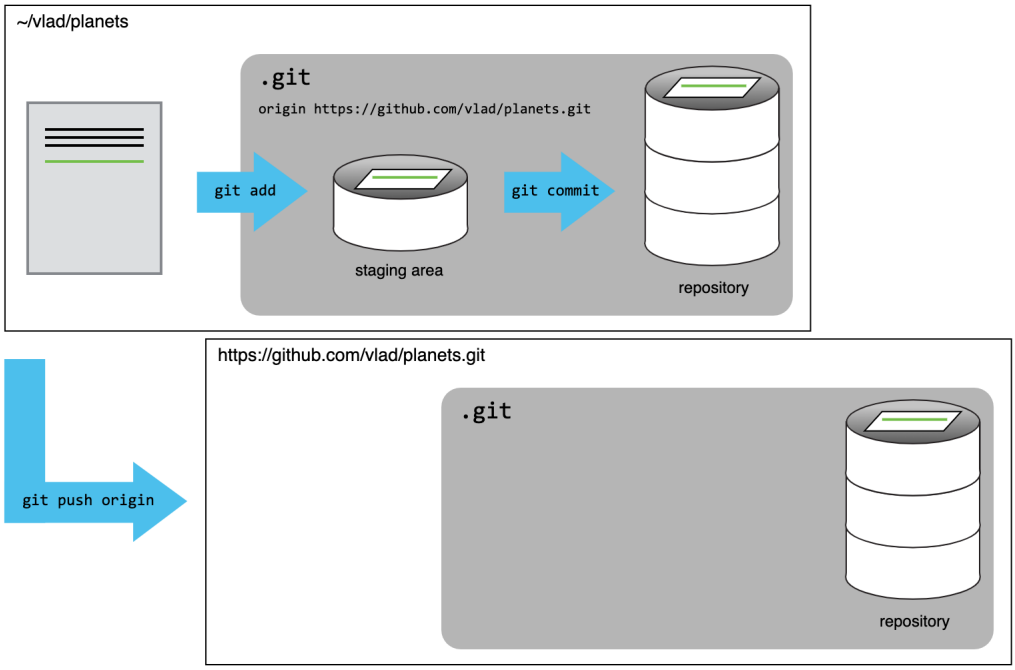


Figure 13: Illustration of pushing changes of files from the local repository to the remote repository on GitHub.

## 2.8 Collaborating

In order to download a copy of the an repository created on your GitHub account to your local computer, one must clone the repository by typing the following code into the command line when in the desired directory (the repository will be created the current directory):

git clone git_repository_url path_to_local_directory

Here is an example of a cloning a repository from GitHub onto a local computer: **git clone https://github.com/vlad/planets.git ~/Desktop/vlad-planets**. If you would like to copy a repository on GitHub over to your GitHub account, then you will need to fork the repository from the others account. By forking, the repository will be copied from the previous owner's account and be placed in your own account under repositories. To fork, go to the main page of repository you would like to fork and in the top right corner should be a button that says fork. From here, you can proceed with the steps above in section 2.7 to sync the remote repository with a local repository. Make sure to give credit to the original owner and developer of the repository!

## 2.9 Conflicts

Conflicts occur when two or more people change the same lines of the same file. That is, if my remote repository changes (i.e. one of my collaborators pushes their local version of the repository to GitHub or I delete or change a file on GitHub in my remote repository which conflicts with my local version) and it is not consistent with my local repository, I will get an error when I try to push my changes to the remote repository. In order to remedy this situation, do the following:

1. Pull the remote version of the remote repository onto you local computer:

   **git pull origin master**

   After the changes from remote branch have been fetched, Git detects that changes made to the local copy overlap with those made to the remote repository, and therefore refuses to merge the two versions to stop us from trampling on our previous work. The conflict is marked in in the affected file beginning with <<<<<<< HEAD.

2. Resolve the Conflict manual by going into the file and removing the conflict.

3. Lastly, push this resolved version to the remote repository to merger the two repositories:

   **git push origin master**

For a full discussion about how to resolve conflicts between your local and remote repositories, refer to the software carpentry website here.

# Appendix

For further information about open science, licensing, citations and hosting with Git, go to the software carpentry website for lessons.

My main goal with this document is to expand and solidifying my knowledge of the Unix shell while supplying a reference document for students of researchers to learn and grow in their knowledge of working on the terminal for data analysis purposes.

# Acknowledgements

Thank you Software Carpentry for supplying most of the content present in this document.

Thank you to the URS Hiestand Scholars program for funding my research throughout the Summer of 2019.

Thank you as well to my mentors Professor Sarah Gille and Bia Villas Bôas for all the guidance and encouragement they has given me throughout my undergraduate research experience.