Bash and Git Workshop Tutorial

Luke Colosi
Software Carpentry Workshop at Scripps Institution of Oceanography
Workshop Date and time : September 23 and 24 from 8:30am to 4:30pm

Contents

1 The Unix Shell: Summary of Basic Commands

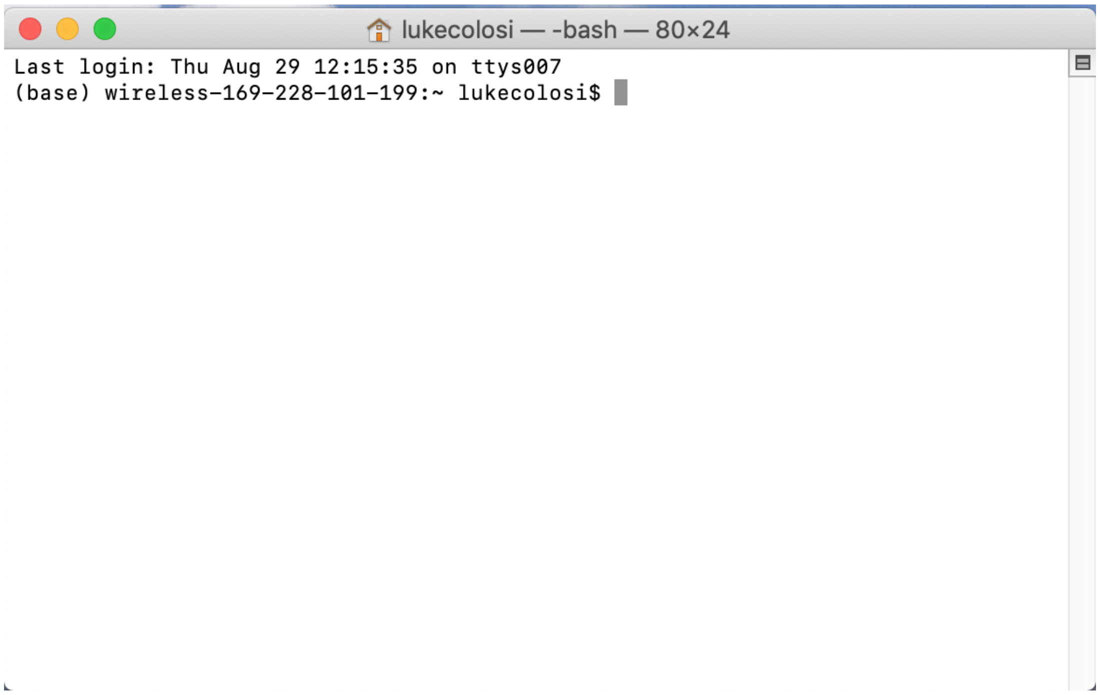
The Bash shell gives you power to do simple jobs more efficiently and in a timely manner in order to streamline researcher's day to day tasks. These jobs can range from moving files, directories, and data locally on your computer to communicating with remote servers to transferring larger data sets and other data management jobs. 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 and typing terminal into the search bar. Up comes that terminal interface and you will see two lines as seen in Figure 1:

Line 1: Last login date and time

Line 2: What directory you are currently in (indicated by the tilde) and what user you are currently login under (indicated by lukecolosi in Figure 1).



```
Last login: Thu Aug 29 12:15:35 on ttys007
(base) wireless-169-228-101-199:~ lukecolosi$
```

Figure 1: 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 benifical 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. First, you cannot use the curse to change the place where you typing. You must scroll over to another location in the command line. Second, 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. Third (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. Fourth, there are many built-in short cuts for the bash shell. Here is link for an extensive list. Fifth, 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.

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 also called files. 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. 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 including 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), etc. 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).

Navigational Bash Commands	
cd path	changes the current working directory
cd ..	move up in the file system hierarchy by one directory
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 is explicit route through the file system which the computer takes in order to preform a command. If the path begins with the root and extends to many directories, 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, ls -la where the flag -la 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 commmand).

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.

File and Directory Manipulation Commands	
cp old new	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 new	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

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 zero or more characters in a filename. For example, *.txt matches all file names that end with .txt. the question mark wildcard (?) matches any single character in a filename. For example, ?.txt matches with a.txt but not with any.txt.

Important features to note about the bash shell. First, the shell does not have a trash bin, so once a file or directory is deleted, the information is gone. Second, file names 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 observation oceanography data 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.

Syntax:

Command1 — Command2 = run command 1, take output from command one and run command 2 with output

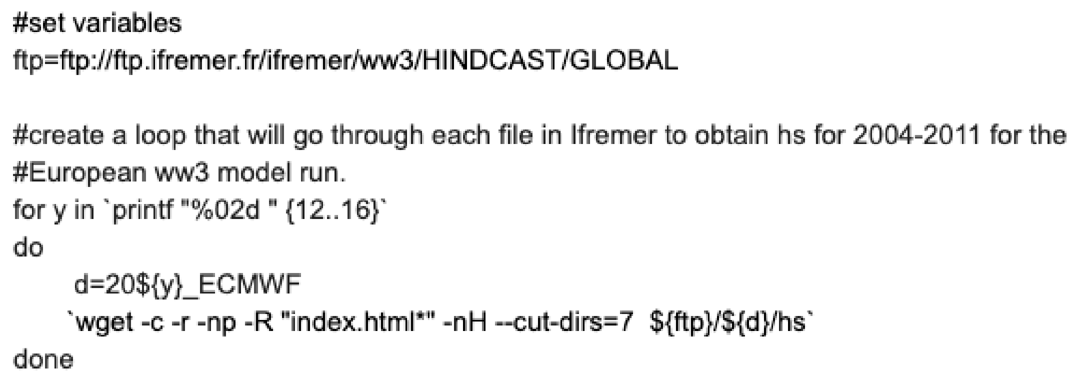
Example: wc -l *.pdb — sort -n — tail -n 1 = outputs the number of lines in multiple files ending with .pdb, sorts the output numerically, and displays the last line

head = display first lines (first 10 as default) of a file tail = display the last part (last 10 lines as default) of a file

To save other output into a new or already existing text file, use: Command ¿¿ filename.txt

1.5 Loops

The Loops focused in this section are for loops. A for loop repeats commands once for every thing in a list. Therefore, every for loop needs two things: a loop variable and a list to loop through. Here is an example of a for loop calling a file and copying it to the local computer.



```
#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 2: For loop example in the Bash Shell

1.6 Text editors and Shell Scripts

1.7 Finding Things

1.8 Using awk

2 Version Control with Git

For a general reference for all Git related topics, use this link. An additional cheatsheet to Git commands can be found 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 track 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 at it. Since all old versions of files are saved, it's always possible to go back in time 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) As 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

Use git config with the –global option to configure a user name, email address, editor, and other preferences once per machine.

Here are some of the necessary configuration sets 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 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
git config –global core.editor "vim"	sets default text editor with Git activities
git config --help	accesses the Git manual

2.3 Creating a Repository

Create a repository under a directory by using **git init**. To check the status of the repository (i.e. what files are being tracked and what modifications have been made, use **git status**.

Nested repositories should be avoided at all cost due to 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 -la in order to print hidden files such as .git.

2.4 Tracking Changes

Begin by creating a README.txt (text), README.md (Markdown), or README.html (HTML) file. 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 files. For a reference to the Markdown syntax, use the link. For a reference to HTML syntax, use the link. To create the readme file, use a text editor such as vim or nano:

vim README.md or nano 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 oin 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 untrack. In order to track a file and to take a snapshot of a file in preparation for versioning, use:

git add filename

To record changes made files in the repository (changes that have been add to the glass which means they are staged with the **git add filename** command and are ready to be committed to the version history) and commit the file snapshot permanently to the version history, 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 with a brief statement. 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**. To look at the project history to see what we have done, use:

git log

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"**. In order to look at the exact changes made to the fileand review then before staging and committing, use:

git diff

To look at the exact changes of stages files, use **git diff –stage**.

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. Fig 3 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.

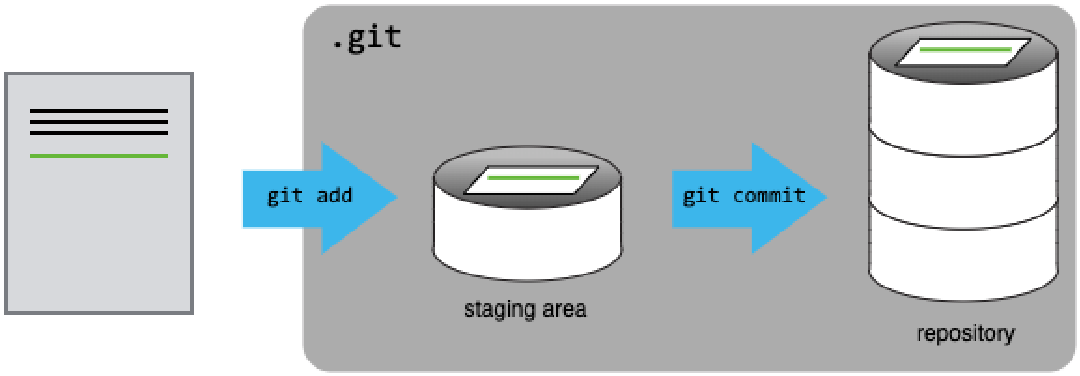


Figure 3: Structure and Mechanics of Git Version History Saving Process

2.5 Exploring History

To show a list of the currently tracked files in the git version history, use:

git ls-tree -r master –name-only

2.6 Ignoring Things

To remove files from git's version history, use the following link. 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 on someone's laptop. The host server where the master copy of files for the repository are help 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 right pulse symbol on the toolbar of your browser interface. This creates a new repository that will show up under your github account. Name the repository. 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". Next a page will show up with three options of how to proceed: 1) create a new repository in the command line; 2) push an existing repository from the command line; 3) import code from another repository. In addition, you can choose between HTTPS and SSH protocols. We will focus on the HTTPS protocol and connecting a local and remote repository. Because we have already created a repository locally, we will choose the second option. This gives us two repositories: the local repository on your computer containing our files and the remote repository on GitHub which is empty. Fig 4 illustrates this situation.

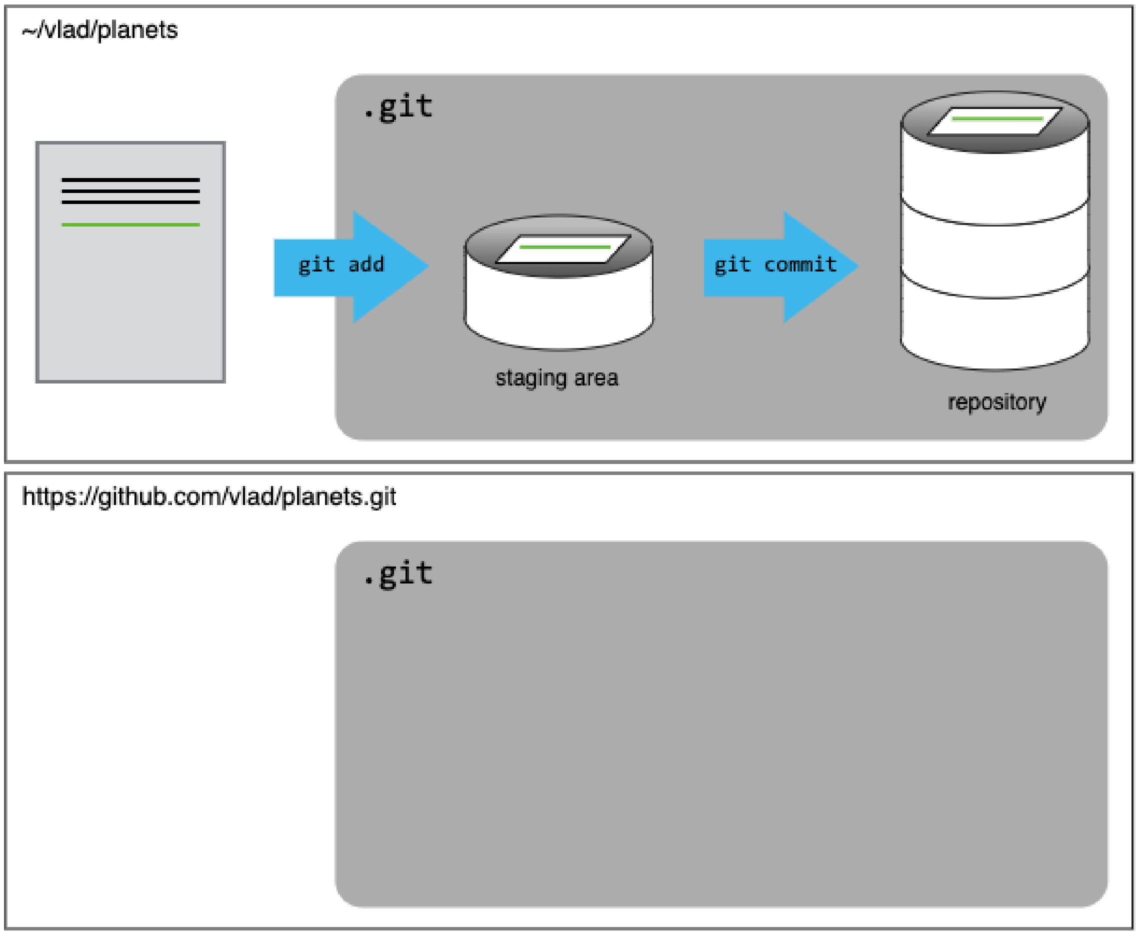


Figure 4: Connecting a local repository with a remote repository on Git Hub: Before connecting repositories stage

Next, go locally to the directory which the Git repository is located and type the following command:

```
git remote add origin https://github.com/lcolosi/SIOC_221A.git
```

to connect 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
```

Once the remote is set up, this command will push the changes from our local repository to the repository on GitHub:

```
git push origin master
```

We can pull changes from the remote repository to the local one as well:

```
git pull origin master
```

2.8 Collaborating

In order to download a copy of the an repository created on your GitHub account to your local machine, 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**. However, if the repository is not originally your our own such that it was not created by you, 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. From here, you can proceed with the steps above to sync the remote repository with a local repository.

2.9 Conflicts

2.10 Open Science

2.11 Licensing

2.12 Citation

2.13 Hosting

Appendix

My summer research project's main goal 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 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.

References