Software Carpentry: Bash, Git and Python (8/11-8/12/22, Virginia Tech)

Table of Contents

Sign in 8/11/22 (Name, Email, Optional Pronouns)

- Nathaniel Porter, ndporter@vt.edu (helper) (he/him/his)
- Matthew Brown, brownm12@vt.edu (instructor) (he/him/his)
- Ashit Harode, ashit02@vt.edu (student) (he/him/his)
- Ernesto Urbaez, uernesto@vt.edu (Graduate Student) (he/him/his)
- Sabrina Amorim, asabrina@vt.edu (Graduate Student) (she/her/hers)
- Brandy Ayesu-Danso, brandyad@vt.edu (Graduate student) (she/her/hers)
- Scott Lucero, dslucero@vt.edu, (Research Faculty) (he/him/his)
- Akhileswar Yanamala, akhileswar@vt.edu (Graduate Student) (he/him/his)
- Festus Animah, aafestus@vt.edu (Graduate student) (he/him/his)
- Benedict Isaac, benedictisaac@vt.edu (Student) (he/him/his)
- Cesar Cardenas, cesarac1975@vt.edu (Graduate student) (he/him/his)
- Aditya Raj, araj@vt.edu (Grad student) (he/him/his)
- Mahyar Arani, mahyar@vt.edu (Graduate student) (he/him/his)
- Pavan Reddy, pavanreddy@vt.edu (Grad student) (He/Him)
- Tosin Ogunmayowa olutosin@vt.edu (Grad Student) (He/Him/His)
- Priyanka Bose, priyanka18@vt.edu (Grad Student) (she/her/hers)

Before we start:

- Follow setup instructions on workshop webpage
- Complete pre-workshop survey
- Sign in

Workshop Overview

- Workshop webpage: https://ndporter.github.io/2022-08-11-vt-swc-python-online/
- Code of Conduct
- Live Coding
- Instructors and Helpers
- No one left behind!
    - Ask questions in chat or verbally
    - Errors are a chance to learn

Automating Tasks with the Unix Shell (8/11 Morning)

Key Links

Setup (data and software)

Notes

- Introducing the Shell
    - Make sure you have the shell and have data downloaded from the setup link
    - Most computing interactions today use graphical interfaces (GUIs) with menus and clicking
    - Some tasks require repeating similar processes over and over (sometimes thousands of time)
    - Shell programming helps to simplify repetitive tasks and automation
    - Shells use only text to interact
    - BASH is the most widely used shell
    - Nelle’s Pipeline - we will be working with data where she needs to automate processing lots of data on a short timeline
    - **ls** : lists files and directories in the current directory
- Navigating Files and Directories
    - **pwd**: print’s location (path) of current working directory
        - Each OS has different directory structures but this is like a filing system for where each file is located
    - ls -F: the -F is an *option*, which in this case provides additional output to distinguish directories from files (with the / at the end)
    - **clear**: clear output from screen

  - Getting help
    - Windows: [command] –help (ex: ls –help)
    - Mac: man [command] (ex: man ls)
      - Type ‘q’ to exit manual
  - Modifying how commands work
    - Options (e.g.ls -F -lh etc) change what the command does or outputs
      - Options start with a -
      - Multiple options can be combined with a single dash (like -lh)
      - Order doesn’t matter
    - Arguments (ls Desktop) provide additional information to a command - like a directory other than the current directory to list the contents of
      - Arguments are specified after the command and any options (COMMAND -OPTIONS ARGUMENT)
  - **cd** : change directory
    - .. and . are special directories - . is the current directory while .. is up one level
    - You can change one level at a time, use multiple separated by a slash, or use the full absolute path (from /users/ or /c/ etc)
    - ‘cd -’ allows you to move to the immediate previous directory (and toggle back and forth between two)
    - ‘cd’ with nothing else or ‘cd ~’ will automatically return you to your home directory
  - Directories and files that begin with a ‘.’ are hidden by default
    - To show in ls, use the -a option
  - Tab-completion allows you to type part of a file or directory name and press tab and it will auto-complete
- Working with Files and Directories
  - mkdir : make one or more new (empty) directories
    - Specify names as arguments (at least one)
    - -p option allows making nested directories together
  - Names
    - Avoid spaces when possible
    - Don’t begin with ‘-’
    - Use only letters/numbers/./-/_ to avoid issues
  - Nano
    - Nano is a simple text editor that is typically available for basic text and keyboard only editing
    - Use CTRL-O to save and follow prompts
    - CTRL-X exits (and prompts to save any unsaved changes)
    - Call with `nano FILENAME`
  - Moving files and directories
    - mv : “move” can be used to change names or directories of a file
    - mv current_file_path/current_file_name new_file_path/new_file_name

    - Move multiple files at once with `mv current_1 current_2 … new_file_path/
      - Notice no file names on the last one because each file will be moved (and keep its name)
  - Copy:
    - For a file: cp current_path_and_file new_path_and_file
    - For a directory: cp -r current_path new_path
      - -r option required for directories
      - Can create a new directory (but copying individual files will not)
  - Remove
    - rm current_path_and_file
    - **Deleting is forever - there’s no recycle bin to recover things you delete**
    - Remove a directory with: `rm -r current_path`
      - Be extra careful here - this will permanently delete everything in the directory, even if it’s your home directory
    - rm -i (interactive) requires you to confirm each file to be deleted
  - Wildcards (pattern matching)
    - Can be used in multiple commands (such as ls)
    - * replaces any one or more character
    - ? replaces exactly one character
- Other sections (not covered today but in curriculum)
  - Pipes and Filters
    - Pipes send the output of one command as input to another
    - Filters help sort or select subsets of data
    - Redirects can send output to a new file or append to the end of an existing file
  - Loops
    - Allow repeating a set of operations on multiple inputs
  - Scripts
    - Allows saving sequence of commands to run repeatedly
    - Can also take inputs that allow you to change options etc
  - Finding things
    - There are multiple ways to search files automatically (filenames or file contents) including special expressions that allow custom strings

NEXT BREAK: 12:30-1:30 (lunch)

Morning 1 feedback (before logging off for lunch)

- What was helpful or went well?
  - Following along with the instructor on my own system really helped me see the result of discussion.

    - Well explained
    - Clear explanation with useful information
    - Matt's pace and speed is really useful!
- What could have been better or was difficult/confusing?
    - More time for scripts and loops would be useful. I think that these are very powerful tools.
    - More time on scripting and looping through file similar to the case study discussed in the beginning of the course would have been great

Plotting and Programming in Python (Part 1- Thursday afternoon)

https://swcarpentry.github.io/python-novice-gapminder/

Python Documentation

- Running and Quitting
    - In Anaconda, access JupyterLab through the Anaconda Navigator
    - JupyterLab runs in your browser and basically keeps a Python session (kernel) open in the background
    - Command mode: Esc (gray)
        - m: Markdown mode (write text)
        - y: Code mode
        - b: Make a new cell below current cell
        - a: Make a new cell above current cell
        - x: Delete the current cell
        - z: Undo
    - Edit mode: Return (blue outline)
        - Write code or text
        - Shift + Return executes contents of the cell
        - Create comment in code: #
    - Closing Jupyter Lab Notebook
        - File -> Shut Down
        - Or From Terminal: Control + c
- Variables and Assignment
    - Use variables to store values
        - age = 42
        - first_name = “Sarah”
        - Use print() to display values
            - When you assign a value to a variable, nothing will print out to the console.
            - If you want to know what that value is, use print: print(first_name)
    - Use an index to get a single character from a string
        - 'helium'[0]

        - Index begins as 0
    - Slice to get a substring
        - string[start:stop]
        - 'sodium'[0:3]
        - Begins at start and goes up to but not including the stop index.
    - Find length of strings
        - len()
- Data Types and Type Conversion
    - Every value has a type
        - Types control what operations (or methods) can be performed on a given value.
        - type(): Find the type of an object.
        - Type dictates what you can do with different objects
        - Compare
            - print(5 - 3)
            - print('hello' - 'h')
    - Can use + and * to operate on strings
        - full_name = “Sara” + “ “ + “Over”
        - ‘=’ * 10
    - Convert between numbers and strings
        - str() to create string
            - print(str(1) + '2')
        - int() to create integer
            - print(1 + int('2'))
    - Can mix integers and floats freely in operations
        - print('half is', 1 / 2.0)
        - print('three squared is', 3.0 ** 2)
    - Division in Python
        - Integer division: 5 // 3
        - Floating point division: 5 / 3
        - Remainder division: 5 % 3
- Built-in Functions and Help
    - A function may take zero or more arguments
    - Every function returns something
        - If the function doesn’t have a useful result to return, it usually returns the special value `None`.
    - max, min, and round functions
        - max and min both work on strings and numbers
        - But they must be given things that can be meaningfully compared. Cannot combine strings and numbers in same call.
        - round() can take two arguments: value and decimal places if desired
            - round(3.712)
            - round(3.712, 1)
    - Method vs function

        - Function: round(3.712)
        - Method: my_string.swapcase()
    - Getting help
        - help(function-name)
        - In Jupyter Notebook place the cursor near where the function is invoked in a cell and hold down Shift + Tab
        - function-name?
        - Also consult the Python Documentation
    - Python errors
        - Syntax error
            - Watch for missing parentheses
        - Runtime error
- Libraries
    - Must import a library module before using it
        - import: Use import to load a library module
            - import math
        - Refer to things from the module as module_name.thing_name
            - math.pi
            - math.cos()
        - Use help to learn about the contents of a library module
            - help(math)
            - A module must be imported/loaded to use help.
    - Import specific items from a library module
        - from math import cos, pi
        - Now we can use pi and cos() without math. notation
        - cos(pi)
    - Create an alias for a library module
        - import math as m
        - m.cos(m.pi)
        - This is commonly used for libraries that are frequently used or have long names.
            - An example is import pandas as pd
- Reading Tabular Data into DataFrames
    - Use the Pandas library to do statistics on tabular data
    - import pandas module
        - import pandas as pd
        - pd is common alias for pandas
    - Read a csv data file with pd.read_csv()
        - data = pd.read_csv('data/gapminder_gdp_oceania.csv')
    - Use index_col to specify that a column's values should be used as row headings
        - data = pd.read_csv('data/gapminder_gdp_oceania.csv', index_col='country')
    - Investigate the aspects of the DataFrame
        - data.info()

  - See the columns of the DataFrame
    - data.columns
    - Note that this is data, not a method. (It doesn’t have parentheses.)
  - Transpose a DataFrame
    - Columns become rows and rows become columns
    - DataFrame.T
  - Get summary statistics about the data
    - data.describe()
  - Inspecting the data
    - Read in longer set of data
      - americas = pd.read_csv('data/gapminder_gdp_americas.csv', index_col='country')
    - Look at the start of the data: americas.head()
      - Only the first three lines: americas.head(n = 3)
    - Look at the end of the data: americas.tail()
      - Only the last three lines: americas.tail(n = 3)
  - Put it all together
    - americas.T.tail(n=3).T
    - Transpose, get tail, and then transpose again
    - Get last three columns
- Pandas DataFrames
  - Subsetting DataFrames
    - By position: DataFrame.iloc[..., ...]
      - First column and first row: data.iloc[0, 0]
    - By label: DataFrame.loc[..., ...]
      - data.loc["Albania", "gdpPercap_1952"]
    - Use : on its own to mean all columns or all rows
      - All columns of a row: data.loc["Albania", :]
      - All rows of a column: data.loc[:, "gdpPercap_1952"]
    - Slices of DataFrames
      - data.loc['Italy':'Poland', 'gdpPercap_1962':'gdpPercap_1972']
      - With slicing, loc is inclusive at both ends, while iloc is inclusive at the beginning and exclusive (does not include) the end point.
  - Use comparisons to select data based on value
    - Which values were greater than 10,000?
      - data > 10000
    - Boolean mask: Turn False into NaN (Not a Number)
      - Defining a boolean mask: Anytime you apply a True/False question to more than one piece of data (vector, dataframe, array), it returns a boolean array of the same size as the data. So when you use the mask, you're just saying keep only the data where the condition is true.
      - mask = data > 10000
      - data[mask]

            - This is useful because NaNs are ignored by operations like max, min, average, etc.
            - data[mask].describe()
    - Group By: split-apply-combine
        - Example of splitting countries in Europe by how often the GDP is above or below the mean GDP.
        - Create a boolean mask
            - mask_higher = data > data.mean()
        - Create a wealth score: How often was each country above or below the mean
            - wealth_score = mask_higher.aggregate('sum', axis=1) / len(data.columns)
- Plotting
    - matplotlib
        - matplotlib tutorial
        - matplotlib is the most widely used scientific plotting library in Python
        - import matplotlib.pyplot as plt
    - Making our first plot
        - x and y values
            - time = [0, 1, 2, 3]
            - position = [0, 100, 200, 300]
        - Make the plot and add axis labels
            - plt.plot(time, position)
            - plt.xlabel('Time (hr)')
            - plt.ylabel('Position (km)')
    - Making a plot with Pandas data
        - data = pd.read_csv('data/gapminder_gdp_oceania.csv', index_col='country')
        - Prepare the data for plotting
            - Need to turn column names into integers
            - Get the years
                - years = data.columns.str.strip('gdpPercap_')
            - Rename the columns
                - data.columns = years.astype(int)
        - Make the plot
            - data.loc['Australia'].plot()
    - Select and transform data, then plot it
        - data.T.plot()
        - plt.ylabel('GDP per capita')
    - Can change the style of plots
        - plt.style.use('ggplot')
        - data.T.plot(kind='bar')
        - plt.ylabel('GDP per capita')
    - Saving plots

        - When run in the same code chunk
            - plt.savefig('my_figure.png')
        - Alternative to save file as variable and then save
            - Create plot
            - Save last figure: fig = plt.gcf()
                - Short for get current figure
            - fig.savefig('my_figure.png')

Afternoon 1 feedback (before logging off for the day)

- What was helpful or went well?
- What could have been better or was difficult/confusing?

Sign in 8/12/22 (Name, Email, Optional Pronouns)

- Jesse Sadler, jrsadler@vt.edu (instructor) (he/him/his)
- Ellie Kohler, elliek@vt.edu (helper) (she/her/hers)
- Ernesto Urbaez, uernesto@vt.edu (Graduate Student) (he/him/his)
- Brandy Ayesu-Danso, brandyad@vt.edu (Graduate Student)(she/her/hers)
- Aditya Raj, araj@vt.edu (Grad student) (he/him/his)
- Benedict Isaac, benedictisaac@vt.edu (Graduate Student)
- Priyanka Bose, priyanka18@vt.edu (Grad Student) (she/her/hers)
- Cesar Cardenas, cesarac1975@vt.edu (Graduate student) (he/him/his)
- Ashit harode, ashit02@vt.edu (he/him)

Plotting and Programming in Python (Part 2 - Friday morning)

https://swcarpentry.github.io/python-novice-gapminder/

Python Documentation

- Questions:
    - When the notebook opened this morning there was unexpectedness.- This is a function of the Jupyter Notebook, as you can edit out of order

- Running and Quitting
    - Command mode: Esc (gray)
        - m: Markdown mode (write text)

    - y: Code mode
    - b: Make a new cell below current cell
    - a: Make a new cell above current cell
    - x: Delete the current cell
    - z: Undo
  - Edit mode: Return (blue outline)
    - Write code or text
    - Shift + Return executes contents of the cell
    - Create comment in code: #
  - Closing Jupyter Lab Notebook
    - File -> Shut Down
    - Or From Terminal: Control + c
- Lists
  - Create a list
    - A list stores many values in a single structure
    - Created with square brackets [ ],
    - items within are separated by commas
      - len(list_name) #how many items/values are in a list
  - Manipulating lists
    - Lists can be sliced and indexed like strings
    - Replacing and appending values with functions
      - list_name.append()   #appending items to the list to make it longer
      - list_name.extend() #adding a list ot a list
      - del list_name[index] #remove items from a list
    - Empty lists contains no values [ ]
  - Characters vs lists
    - Character strings are immutable- cannot replace individual characters within a string.
    - Lists and characters can both be indexed
- For Loops
  - For loop tutorial and syntax

    ```
    for __ in ___:
       command
    ```

    - The loop executes commands once for each value in a collection
    - A for loop is made of collection, loop variable, body
    - Formatting-
      - must end with a colon
      - body must be indented
        - Body can contain many statements.
      - Loop variable names can be anything
  - Range- iterate over a sequence of numbers

    ```
    for ___ in range (   ,   ):
       command
    ```

    - a range is not a list, and does not act the same way

    - Accumulator- turn many values into one
        - Initialize by creating a variable that equals 0, an empty string, or an empty list.
- Conditionals
    - if statements -
        - Controls whether a block of code is executed
            - if
                - if it is true, execute command
                - goes in the beginning
            - else
                - follows if.
                - If a condition is not true, execute command
            - elif
                - short for else if,
                - use when you want to provide several alternative choices
                - goes in between if and else
                - can have as many elifs as you want
        - Often used inside loops
        - Ordering matters- the branches of a conditional are tested in order of the written command.
            - Conditions are tested once (until it’s true) and in order
            - Variables can evolve within a loop
    - Compound relations
        - and
            - If a statement hits every condition
        - or
            - If a statement has one of the conditions true

- Looping Over Data Sets
    - For loops can read in several files of data
        - Can call files individually
    - Glob tutorial
        - Glob means matching a set of files with a pattern
            - * match zero or more characters
            - ? match exactly 1 character
import glob
Glob.glob(“*.txt”) #matches all files in the directory that has a name ending in .txt
        - Use glob and for to process batches of files
for filename in glob.glob (‘folder/*.csv):
    data = pd.read_csv(filename)
- Writing Functions
    - Reuse code- if you write the same code more than twice, you may want to think about writing a function. Iterative process

  - Define a function
    - *def function_name(parameters):*
      command / block of code

    - Defining a function does not run it
  - Arguments
    - Matched to parameters in definition
    - Functions are most useful when they can operate on different data.
    - Specify *parameters* when defining a function.
      - These become variables when the function is executed.
      - Are assigned the arguments in the call (i.e., the values passed to the function).
      - If you don't name the arguments when using them in the call, the arguments will be matched to parameters in the order the parameters are defined in the function.
  - Return command
    - return()
    - Give the value back to the caller in a function.
    - Saves to an object/ variable

Afternoon 2 feedback (before logging off for lunch)

- What was helpful or went well?
- What could have been better or was difficult/confusing?

Version Control with Git (Friday afternoon)

https://swcarpentry.github.io/git-novice/index.html

- Automated Version Control
  - Why you should use version control.
  - Version control systems start with a base version of the document and then record changes you make each step of the way.
    - Power of separating changes from the document itself.
    - Opens ability for multiple people to make changes at the same time.
  - Version control provides
    - Record of changes, of commits
    - Complete history of commits of a project and their metadata make up a repository.
- Setting up Git
  - Info needed to set up Git

    - our name and email address
      - `git config --global user.name "Vlad Dracula"`
      - `git config --global user.email "vlad@tran.sylvan.ia"`
    - preferred text editor
      - `git config --global core.editor "nano -w"`
    - and that we want to use these settings globally (i.e. for every project).
    - Default branch name
      - `git config --global init.defaultBranch main`
  - Check your settings
    - `git config --list`
  - Get help on configuration commands
    - `git config -h` or `git config --help`
- Creating a Repository
  - Create a repository
    - `cd ~/Desktop`
    - `mkdir planets`
  - Initiate git repository
    - `git init`
    - `.git` file
  - Make sure you are on branch main
    - Check branch name: `git branch --show-current`
    - Create and move to main branch: `git checkout -b main`
  - Check everything
    - `git status`
- Tracking Changes
  - Create mars.txt
  - `git add`
  - `git commit -m ""`
    - Writing good commit messages.
  - `git log`
  - Make another change
  - `git diff`
  - Go over two-step process of committing
    - Staging area with `git add`
    - Actual commit with `git commit`
    - Advice not to use `git commit -a` but to commit files manually.
  - Make another change
    - Stage changed file
    - `git diff --staged`
  - `git log`
    - Limit size of log: `git log -1`
    - `git log --oneline`
    - `git log --oneline --graph`
  - git and directories

    - Git does not track directories on their own, only files within them.
      - Create a directory and run `git status`
    - Add files in a directory with `git add directory-name`
  - Committing multiple files
    - Make change to mars.txt
    - Create venus.txt:
      - `echo "Venus is a nice planet and I definitely should consider it as a base." > venus.txt`
    - `git add mars.txt venus.txt`
    - `git commit`
- Exploring History
  - `HEAD`
  - Exploring history
    - `git diff HEAD mars.txt` makes explicit `git diff` because you are doing `diff` based on the `HEAD`.
    - dff` with previous commits
      - `git diff HEAD~1 mars.txt`
    - `git show`
      - Shows the changes made at an older commit as well as the commit message.
    - Use of commit ID
      - `git diff 8cc62aa84be902807ee058493e689fda64843829 mars.txt`
      - Way too long
    - Use of SHA (first 7 characters of ID)
      - `git diff 8cc62aa mars.txt`
  - Restoring history
    - Restore modified document to `HEAD`
      - `git checkout HEAD mars.txt`
      - Also works if changes are staged.
    - Restore document to previous commit
      - `git checkout c0881d2 mars.txt`
      - This places changes in the staging area.
      - Check with `git status`
    - Go back to `HEAD`
      - `git checkout HEAD mars.txt`
  - Detached `HEAD`
  - It is important to remember that we must use the commit number that identifies the state of the repository before the change we're trying to undo.
  - Explore history of one document
    - `git log mars.txt`
    - `git log --patch mars.txt`: See both commit messages and differences.
- Ignoring Things
  - `nano .gitignore`
    - Create and add to `.gitignore`

        - Add and commit `.gitignore` so that others can have the same file.
    - Using `.gitignore` helps us avoid accidentally adding files to the repository that we don’t want to track.
        - Force adding ignored files: `git add -f a.dat`
        - Show ignored files: `git status --ignored`
    - Including specific files
        - `!final.dat `
- Remotes in GitHub
    - 1. Create a remote repository
        - Login to GitHub
        - Create a new repository with the same name as your git repository
        - Do not add README, .gitignore, or license.
        - This essentially creates a git repository on GitHub's servers.
    - 2. Connect local to remote repository
        - `git remote add origin git@github.com:vlad/planets.git`
        - Check with `git remote -v`
    - 3. Create an SSH key pair
        - Check if key pairs already exist on computer
            - `ls -al ~/.ssh`
        - Create the keys
            - `ssh-keygen -t ed25519 -C "vlad@tran.sylvan.ia"`
                - `-t`: specifies which algorithm to use
                - `-C`: attaches a comment
            - Hit ⏎ to use default file.
            - Enter passphrase
        - Check that key pairs were created
            - `ls -al ~/.ssh`
        - Copy the public key to GitHub
            - Get public key
                - `cat ~/.ssh/id_ed25519.pub`
            - Go to GitHub
                - Settings -> SSH and GPG Keys -> New SSH key
                - Add name and copy public key
            - Connect
                - `ssh -T git@github.com`
    - 4. Push local changes to a remote
        - `git push origin main`
        - Alternative to use `-u` to set origin as upstream (same as `--set-upstream-to`)
            - `git push -u origin main`
    - 5. Pull changes
        - `git pull origin main`
- Collaborating
- Conflicts