Developing Fundamental AI Programming Skills for Drug Discovery

Texas Advanced Computing Center

Dec 09, 2021

CONTENT:

This guide contains the content for **Jupyter notebooks and portal access at TACC** and **Python fundamentals and foundations for AI**. The objectives of these sections are to introduce the use of Jupyter notebooks at TACC, and to introduce the most fundamental Python skills that are necessary for domain scientists to be successful in AI coding projects.

CHAPTER

ONE

ONBOARDING TO TACC

The Texas Advanced Computing Center (TACC) at UT Austin designs and operates some of the world's most powerful computing resources. The center's mission is to enable discoveries that advance science and society through the application of advanced computing technologies.

We will be using cloud resources at TACC as our development environment. We will access the cloud resources via our SSH clients and TACC account credentials, as well as through a visualization portal.

Attention: Everyone please apply for a TACC account now using this link. If you already have a TACC account, you can just use that. Send your TACC username to wallen [at] tacc [dot] utexas [dot] edu as soon as possible.

1.1 About TACC

TACC is a Research Center, part of UT Austin, and located at the JJ Pickle Research Campus.

TACC at a Glance

Other TACC Services

- Portals and gateways
- Web service APIs
- Rich software stacks
- Consulting
- Curation and analysis
- Code optimization
- Training and outreach
- => Learn more

TACC Partnerships

- NSF: Leadership Class Computing Facility (LCCF)
- NSF: Extreme Science and Engineering Discovery Environment (XSEDE)
- UT Research Cyberinfrastructure (UTRC)
- TX Lonestar Education and Research Network (LEARN)
- Industry, STAR Program
- International, The International Collaboratory for Emerging Technologies

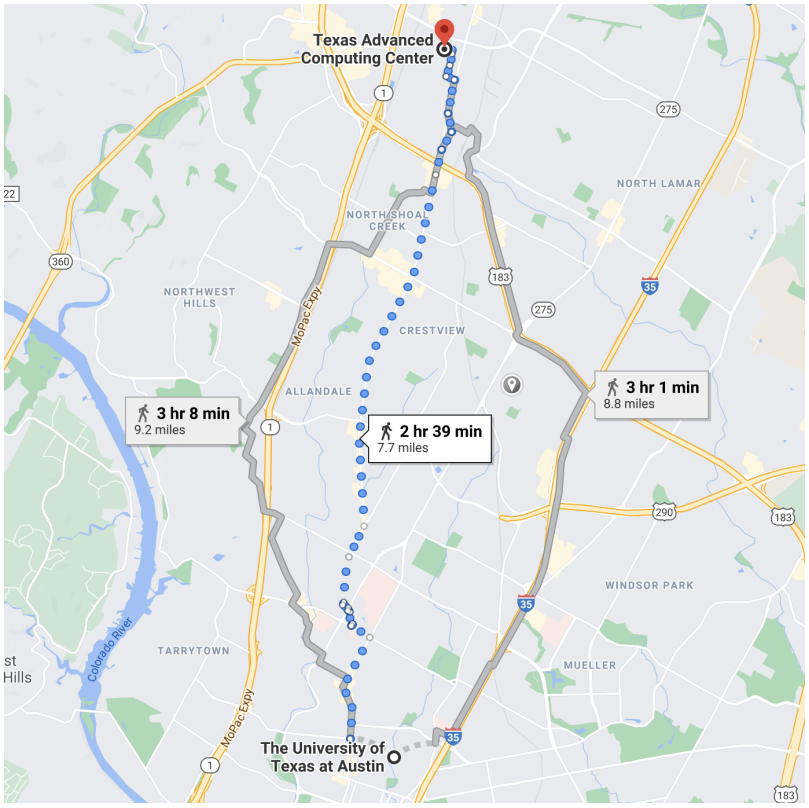
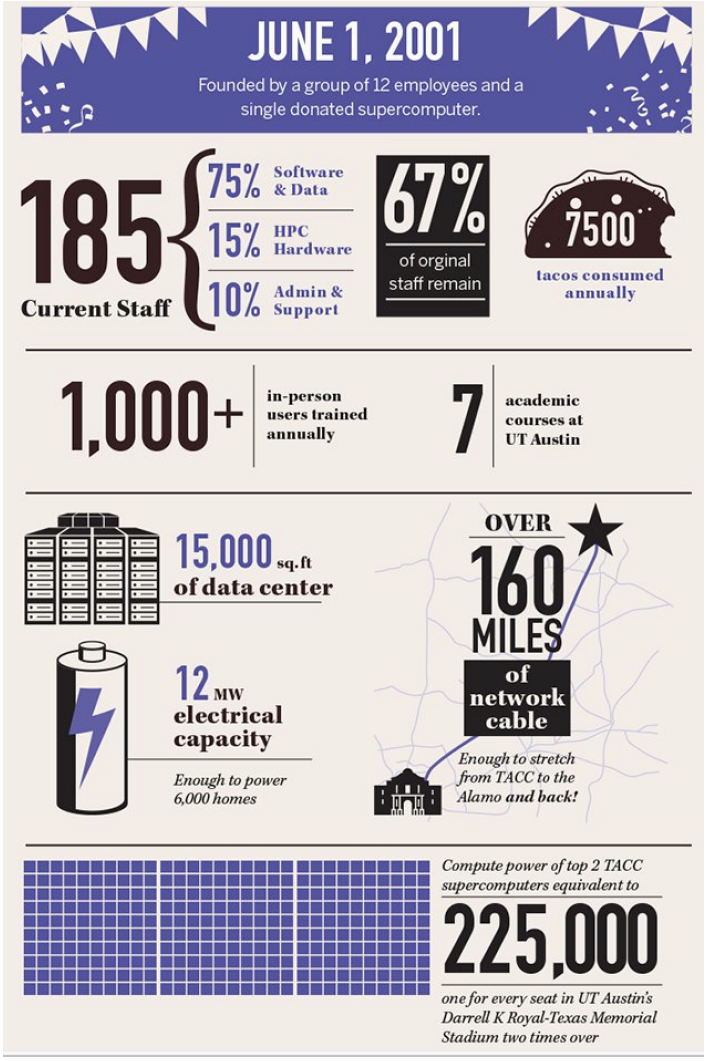
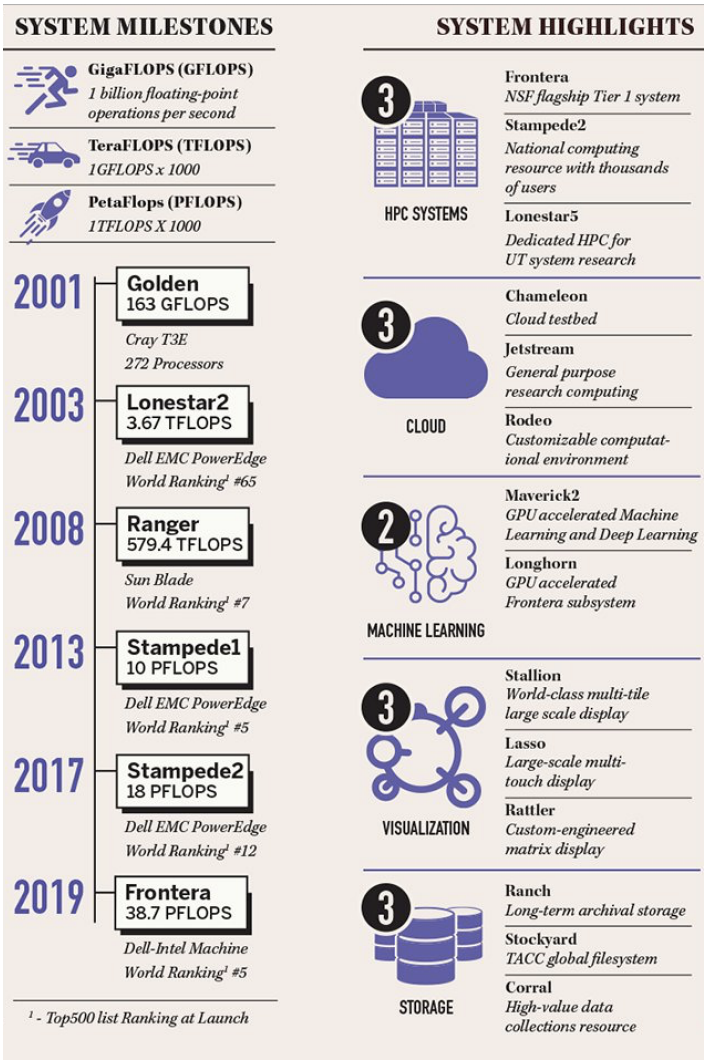


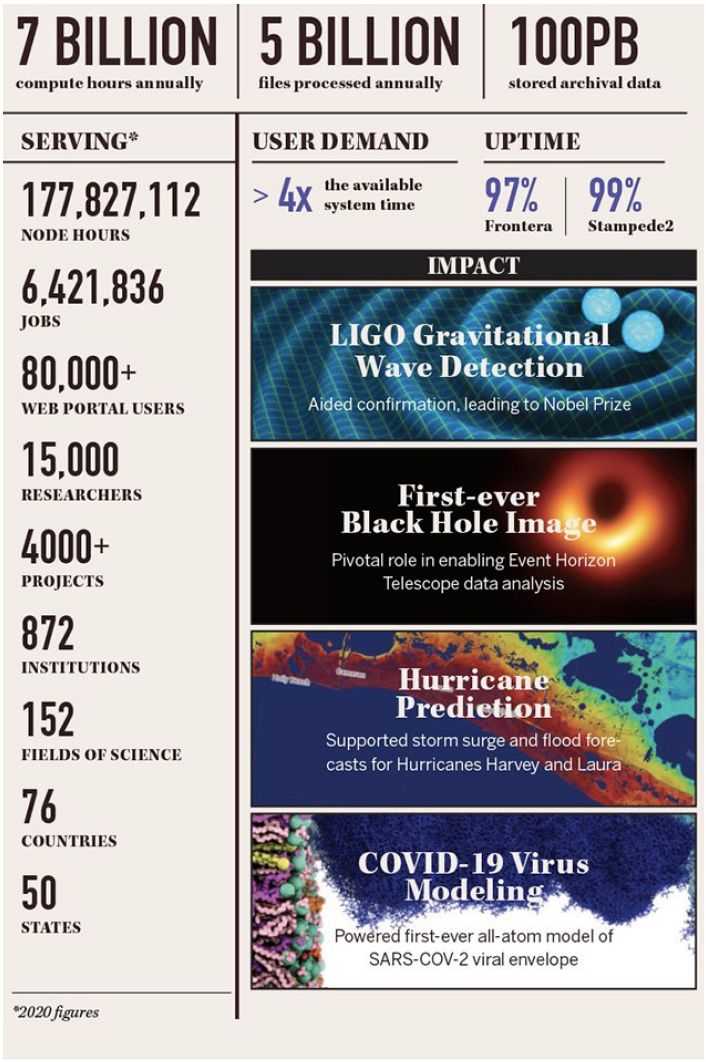
Fig. 1: A short 7.7 mile walk from main campus!

Fig. 2: One of two TACC buildings located at JJ Pickle.

Fig. 3: A tall guy standing among taller Frontera racks.




SYSTEM MILESTONES
GigaFLOPS (GFLOPS)
1 billion floating-point operations per second
TeraFLOPS (TFLOPS)
1GFLOPS x 1000
PetaFlops (PFLOPS)
1TFLOPS X 1000
2001
Golden
163 GFLOPS
Cray T3E
272 Processors
2003
Lonestar2
3.67 TFLOPS
Dell EMC PowerEdge
World Ranking[1] #65
2008
Ranger
579.4 TFLOPS
Sun Blade
World Ranking[1] #7
2013
Stampede1
10 PFLOPS
Dell EMC PowerEdge
World Ranking[1] #5
2017
Stampede2
18 PFLOPS
Dell EMC PowerEdge
World Ranking[1] #12
2019
Frontera
38.7 PFLOPS
Dell-Intel Machine
World Ranking[1] #5
[1] - Top500 list Ranking at Launch
SYSTEM HIGHLIGHTS
3
HPC SYSTEMS
Frontera
NSF flagship Tier 1 system
Stampede2
National computing resource with thousands of users
Lonestar5
Dedicated HPC for UT system research
3
CLOUD
Chameleon
Cloud testbed
Jetstream
General purpose research computing
Rodeo
Customizable computational environment
2
MACHINE LEARNING
Maverick2
GPU accelerated Machine Learning and Deep Learning
Longhorn
GPU accelerated Frontera subsystem
3
VISUALIZATION
Stallion
World-class multi-tile large scale display
Lasso
Large-scale multi-touch display
Rattler
Custom-engineered matrix display
3
STORAGE
Ranch
Long-term archival storage
Stockyard
TACC global filesystem
Corral
High-value data collections resource


7 BILLION
compute hours annually
5 BILLION
files processed annually
100PB
stored archival data
SERVING*
177,827,112
NODE HOURS
6,421,836
JOBS
80,000+
WEB PORTAL USERS
15,000
RESEARCHERS
4000+
PROJECTS
872
INSTITUTIONS
152
FIELDS OF SCIENCE
76
COUNTRIES
50
STATES
*2020 figures
USER DEMAND
> 4x the available system time
UPTIME
97% Frontera
99% Stampede2
IMPACT
LIGO Gravitational Wave Detection
Aided confirmation, leading to Nobel Prize
First-ever Black Hole Image
Pivotal role in enabling Event Horizon Telescope data analysis
Hurricane Prediction
Supported storm surge and flood forecasts for Hurricanes Harvey and Laura
COVID-19 Virus Modeling
Powered first-ever all-atom model of SARS-COV-2 viral envelope

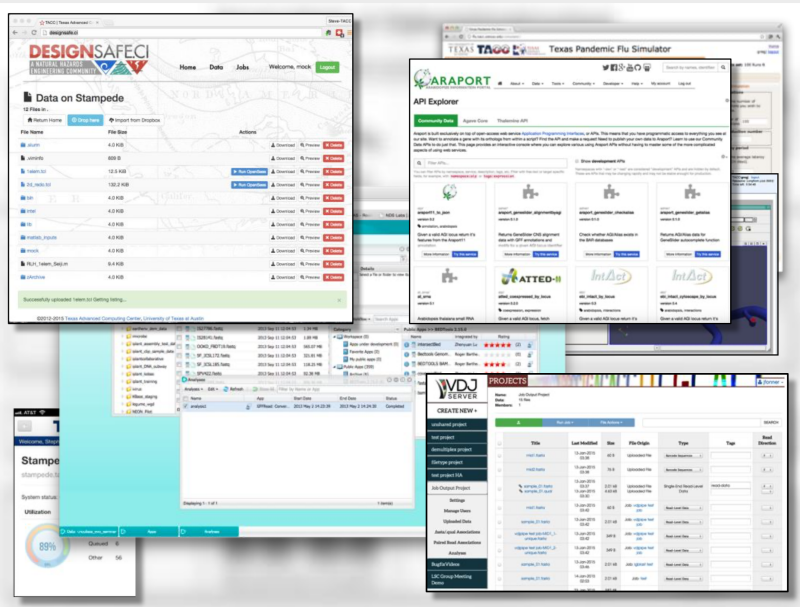


Fig. 4: Snapshot of a few of TACC's portal projects.

- => Learn more

Attention: Did you already e-mail me your TACC username?

1.2 Before We Continue

Using your SSH client, please try to log in to Longhorn. Make sure to use your own username in place of `username`:

```
[local]$ ssh username@longhorn.tacc.utexas.edu
To access the system:

1) If not using ssh-keys, please enter your TACC password at the password prompt
2) At the TACC Token prompt, enter your 6-digit code followed by <return>.

Password:
TACC Token Code:
Last login: Mon Dec  6 15:36:46 2021 from 192.168.64.11
--------------------------------------------------------------------------------
                    Welcome to the Longhorn Supercomputer
      Texas Advanced Computing Center, The University of Texas at Austin
--------------------------------------------------------------------------------

              ** Unauthorized use/access is prohibited. **

If you log on to this computer system, you acknowledge your awareness
of and concurrence with the UT Austin Acceptable Use Policy. The
University will prosecute violators to the full extent of the law.

TACC Usage Policies:
```

(continues on next page)

(continued from previous page)

```
http://www.tacc.utexas.edu/user-services/usage-policies/
_____________________________________________________________________________

Welcome to Longhorn, *please* read these important system notes:

--> Longhorn user documentation is available at:
       https://portal.tacc.utexas.edu/user-guides/longhorn

 |\                                                                     /|
 | \___________________________________.------._____________________________/ |
 '.__________________________________                 ___________________________________.'
     _                          <_/|        |\_>
    | |      ___   __   _   ___     |      |    _    _   ___   ____   __   _
    | |     / _ \|  \| |/ __)     |    |    | |_| |/ _ \| .. \|  \| |
    | |___| (_) | \ \ | (_--    (      )   |  _  | (_) | '' /| \ \ |
    |_____|\___/|_|\__|\___/     '---'    |_| |_|\___/|_|\_\|_|\__|

---------------------- Project balances for user username ----------------------
| Name           Avail SUs     Expires | Name           Avail SUs     Expires |
| ASC21018           12000  2022-06-30 | SD2E-Communit       5548  2021-12-31 |
| TACC-SCI          221130  2025-06-30 | TRA21002            2040  2021-12-31 |
------------------------- Disk quotas for user username ------------------------
| Disk         Usage (GB)     Limit    %Used   File Usage       Limit   %Used |
| /work             436.5    1024.0    42.63      1445104     3000000   48.17 |
| /home              28.1      40.0    70.26       192235      512000   37.55 |
| /scratch          571.3       0.0     0.00     10273017           0    0.00 |
-----------------------------------------------------------------------------

login2.longhorn(1000)$       # success!
```

Also, please try to log in to this Vis Portal using your TACC username and password:

https://vis02.tacc.utexas.edu/

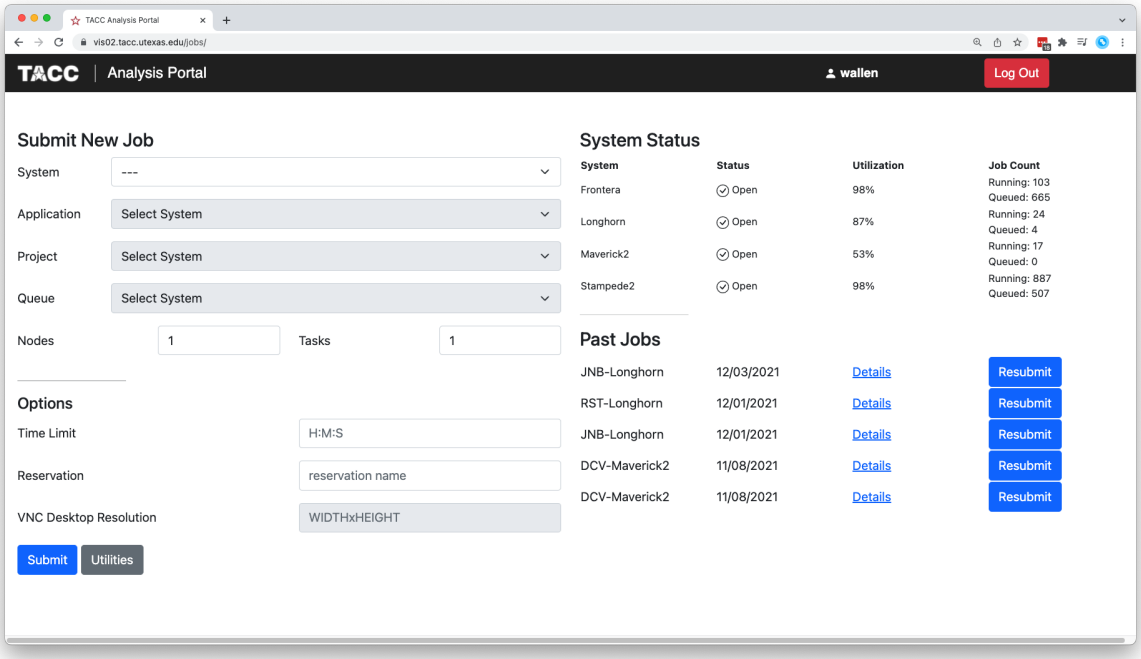


Fig. 5: Successful login.

CHAPTER

TWO

JUPYTER NOTEBOOKS

2.1 What are Jupyter Notebooks?

Jupyter Notebooks are a web-based, interactive computing tool for capturing the whole computation process: developing, documenting, and executing code, as well as communicating the results. They support interactive data science and scientific computing across all programming languages, including Python.

2.2 How do Jupyter Notebooks Work?

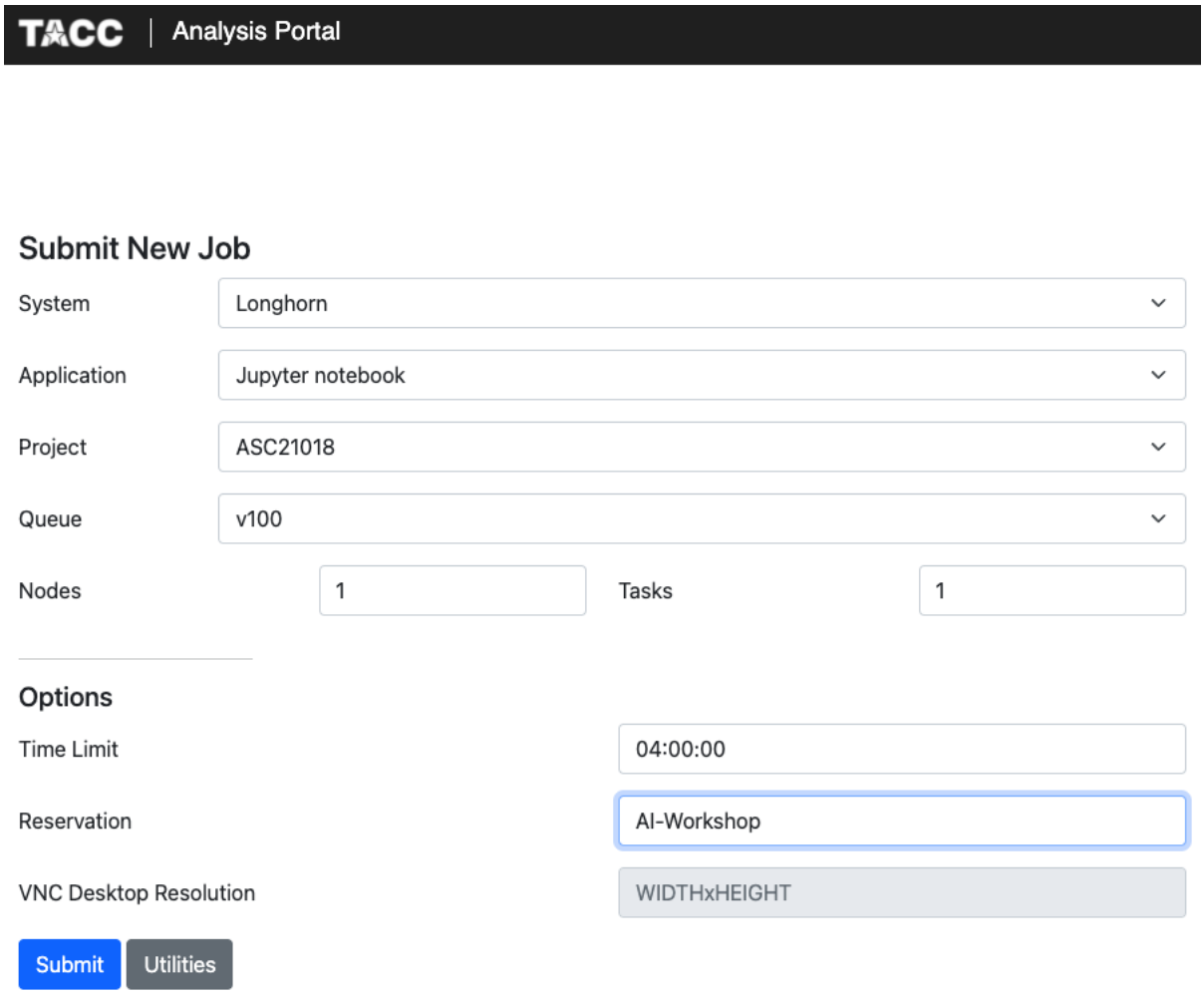
An open notebook has exactly one interactive session connected to a kernel which will execute code sent by the user and communicate back results. This kernel remains active if the web browser window is closed, and reopening the same notebook from the dashboard will reconnect the web application to the same kernel. The kernel's state, including imported libraries and declared variables, persists between cells.

2.3 How do You Access Jupyter Notebooks?

1. The Jupyter website allows you to download Jupyter Notebook and/or try it in your browser.
2. The TACC Analysis Portal allows TACC users to run Jupyter through your browser on Frontera, **Longhorn**, Maverick2, or Stampede2, so long as you have an allocation on that system.
3. The TACC DesignSafe Portal allows anyone with a TACC account to use Jupyter. (Log in, choose Workspace => Tools and Applications => Jupyter)
4. TACC provides small, short lived Jupyter notebooks to anyone with a TACC account via the TACC Cloud.
5. Public Jupyter notebooks are available through sites like Google Colaboratory.
6. Service exist to share notebooks in a browser including Binder.

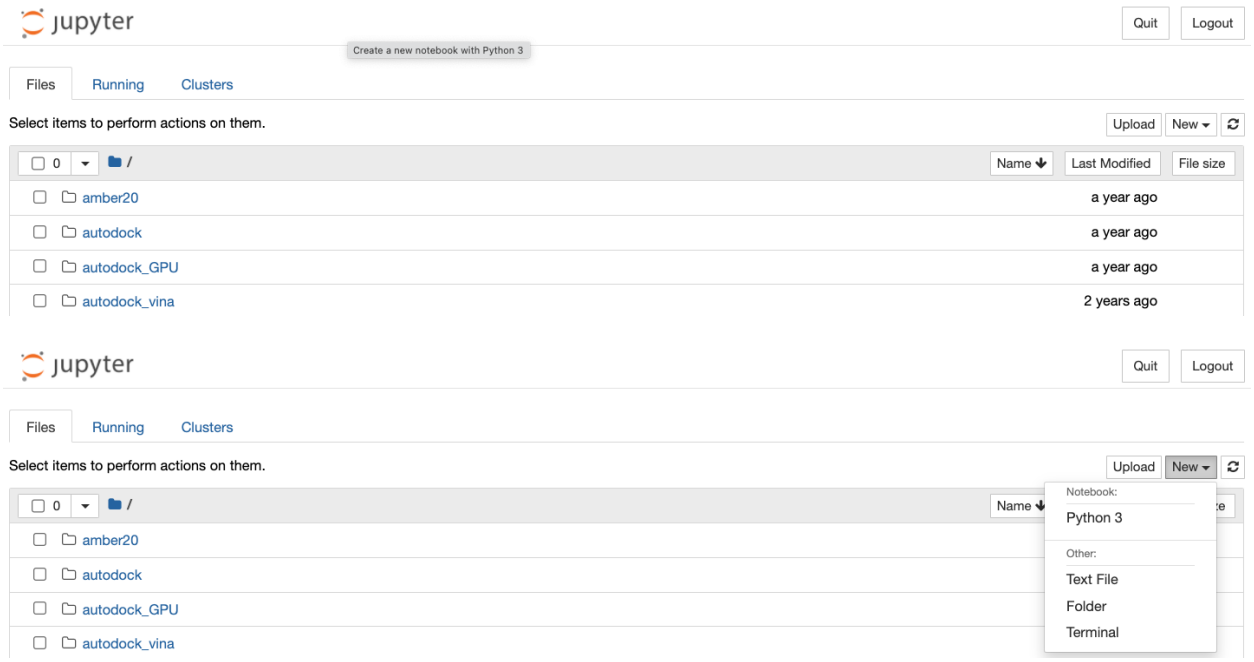
2.4 Interacting with Jupyter Notebooks

First, choose one of the methods above and start a new Jupyter Notebook. I will be following method #2 - the TACC Analysis Portal. For this workshop, use the following settings:



The original, and more common "tree" interface to Jupyter will display a file browser (in this case to files in your Longhorn /home directory):

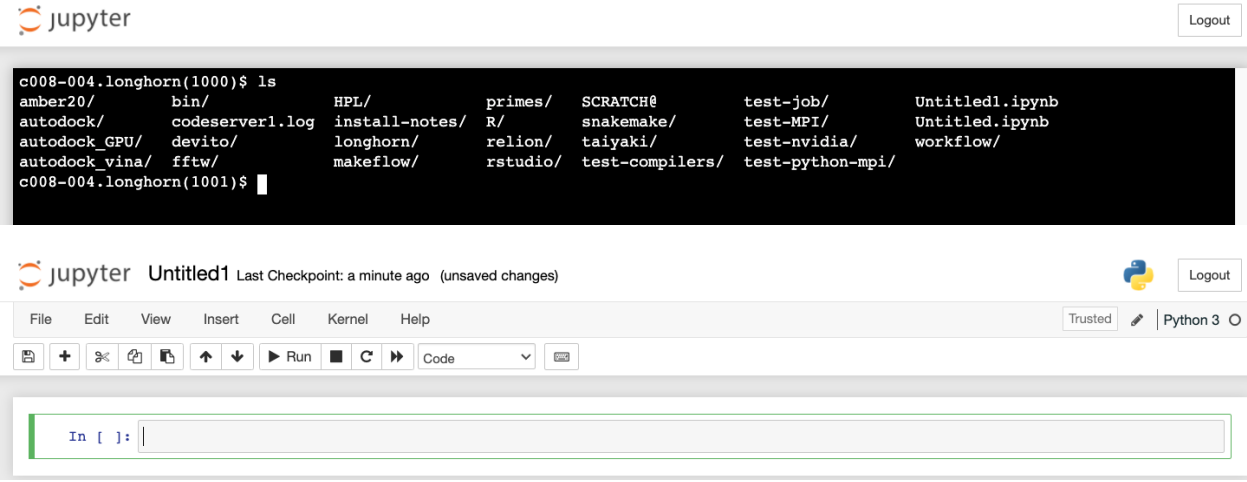
New Python3 kernels and Terminals can be launched by clicking the "New" button on the right side:



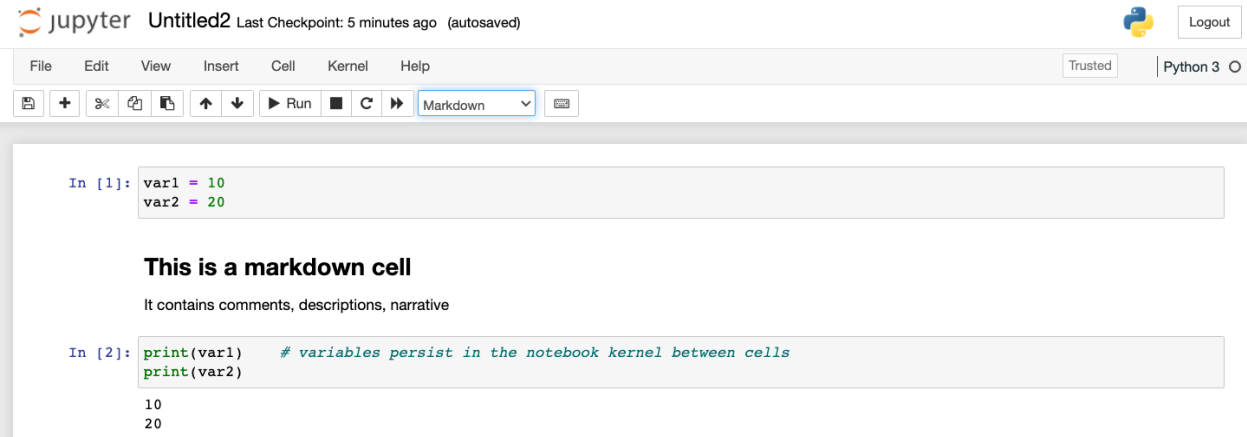
Running kernels can be accessed through the "Running" tab on the top. By default, the Jupyter Notebooks will be saved in your /home directory with a `.ipynb` extension. (Re-opening the notebook will open all the cells, and restore the state of the kernel - more on this later):

The terminal is a standard, fully-functional terminal. Very useful for debugging and interacting with files / directories. For example, you can easily `wget` a data set or unzip a file:

The Jupyter notebook consists of a toolbar (top) and an unlimited number of cells (bottom):



Cells can be either "code cells" or "markdown cells". Code cells allow you to enter and run code. Markdown cells use the markdown markup language to comment on or narrate what is happening in the notebook:



2.5 Miscellaneous Tips and Tricks

Shortcuts:

- Shift+Enter: run cell
- Ctrl+Enter: run cell in place
- Alt+Enter: run cell, insert below
- Esc / Enter: toggle between command and edit mode

Run a command within a Jupyter notebook (prefix with `!`):

```
In[]: ! pip list
In[]: ! pip install --user names
```

The file browser is not (by default) aware of your /work or /scratch directories on Longhorn. The easiest thing to do is open up a terminal and make symbolic links to your /work and /scratch directories in your /home directory

```
[longhorn]$ cd
[longhorn]$ pwd
/home/012345/username
[longhorn]$ ln -s $WORK WORK
[longhorn]$ ln -s $SCRATCH SCRATCH
```

2.6 Exercise

Make a copy of a previously-prepared demo notebook in your /home directory. Open the notebook with Jupyter. Execute the cells within and think about why it is organized the way it is. To copy the notebook:

```
[longhorn]$ cd     # cd to /home directory
[longhorn]$ cp /scratch/03439/wallen/AI-Drug-Discovery/notebook_demo.ipynb .
```

Once you run each cell, save the notebook with a new name. Re-open the original notebook as well as the new notebook. Does saving the notebook save the state of the kernel as well?

CHAPTER

THREE

LINUX REFRESHER

After working through this material, attendees should be able to:

- Describe basic functions of essential Linux commands
- Use Linux commands to navigate a file system and manipulate files
- Transfer data to / from a remote Linux file system
- Edit files directly on a Linux system using a command line utility (e.g. vim, nano, emacs)

Topics covered in this module include:

- Creating and navigating folders (`pwd`, `ls`, `mkdir`, `cd`, `rmdir`)
- Creating and manipulating files (`touch`, `rm`, `mv`, `cp`)
- Looking at the contents of files (`cat`, `more`, `less`, `head`, `tail`, `grep`)
- Network and file transfers (`hostname`, `whoami`, `logout`, `ssh`, `scp`, `rsync`)
- Text editing with vim (insert mode, normal mode, navigating, saving, quitting)

3.1 Log in to Longhorn

To log in to the Longhorn cluster, follow the instructions for your operating system or ssh client below.

Mac / Linux

```
Open the application 'Terminal'
ssh username@longhorn.tacc.utexas.edu
(enter password)
(enter 6-digit token)
```

Windows

```
Open the application 'PuTTY'
enter Host Name: longhorn.tacc.utexas.edu
(click 'Open')
(enter username)
(enter password)
(enter 6-digit token)
```

If you can't access Longhorn yet, a local or web-based Linux environment will work for this guide. However, you will need to access Longhorn for future materials.

Try this Linux environment in a browser.

3.2 Creating and Navigating Folders

On a Windows or Mac desktop, our present location determines what files and folders we can access. I can "see" my present location visually with the help of the graphic interface - I could be looking at my Desktop, or the contents of a folder, for example. In a Linux command-line interface, we lack the same visual queues to tell us what our location is. Instead, we use a command - `pwd` (print working directory) - to tell us our present location. Try executing this command in the terminal:

```
$ pwd
/home/03439/wallen
```

This home location on the Linux filesystem is unique for each user, and it is roughly analogous to C:\Users\username on Windows, or /Users/username on Mac.

To see what files and folders are available at this location, use the `ls` (list) command:

```
$ ls
```

I have no files or folders in my home directory yet, so I do not get a response. We can create some folders using the `mkdir` (make directory) command. The words 'folder' and 'directory' are interchangeable:

```
$ mkdir folder1
$ mkdir folder2
$ mkdir folder3
```

```
$ ls
folder1 folder2 folder3
```

Now we have some folders to work with. To "open" a folder, navigate into that folder using the `cd` (change directory) command. This process is analogous to double-clicking a folder on Windows or Mac:

```
$ pwd
/home/03439/wallen/
$ cd folder1
$ pwd
/home/03439/wallen/folder1
```

Now that we are inside `folder1`, make a few sub-folders:

```
$ mkdir subfolderA
$ mkdir subfolderB
$ mkdir subfolderC
$ ls
subfolderA subfolderB subfolderC
```

Use `cd` to Navigate into `subfolderA`, then use `ls` to list the contents. What do you expect to see?

```
$ cd subfolderA
$ pwd
/home/03439/wallen/folder1/subfolderA
$ ls
```

There is nothing there because we have not made anything yet. Next, we will navigate back to the home directory. So far we have seen how to navigate "down" into folders, but how do we navigate back "up" to the parent folder? There are different ways to do it. For example, we could specify the complete path of where we want to go:

```
$ pwd
/home/03439/wallen/folder1/subfolderA
$ cd /home/03439/wallen/folder1
$ pwd
/home/03439/wallen/folder1/
```

Or, we could use a shortcut, `..`, which refers to the **parent folder** - one level higher than the present location:

```
$ pwd
/home/03439/wallen/folder1
$ cd ..
$ pwd
/home/03439/wallen
```

We are back in our home directory. Finally, use the `rmdir` (remove directory) command to remove folders. This will not work on folders that have any contents (more on this later):

```
$ mkdir junkfolder
$ ls
folder1 folder2 folder3 junkfolder
$ rmdir junkfolder
$ ls
folder1 folder2 folder3
```

Before we move on, let's remove the directories we have made, using `rm -r` to remove our parent folder `folder1` and its subfolders. The `-r` command line option recursively removes subfolders and files located "down" the parent directory. `-r` is required for non-empty folders.

```
$ rm -r folder1
$ ls
folder2 folder3
```

Which command should we use to remove `folder2` and `folder3`?

```
$ rmdir folder2
$ rmdir folder3
$ ls
```

3.3 Creating and Manipulating Files

We have seen how to navigate around the filesystem and perform operations with folders. But, what about files? Just like on Windows or Mac, we can easily create new files, copy files, rename files, and move files to different locations. First, we will navigate to the home directory and create a few new folders and files with the `mkdir` and `touch` commands:

```
$ cd     # cd on an empty line will automatically take you back to the home directory
$ pwd
/home/03439/wallen
$ mkdir folder1
$ mkdir folder2
$ mkdir folder3
$ touch file_a
$ touch file_b
$ touch file_c
```

(continues on next page)

(continued from previous page)

```
$ ls
file_a  file_b  file_c  folder1  folder2  folder3
```

These files we have created are all empty. Removing a file is done with the `rm` (remove) command. Please note that on Linux file systems, there is no "Recycle Bin". Any file or folder removed is gone forever and often un-recoverable:

```
$ touch junkfile
$ rm junkfile
```

Moving files with the `mv` command and copying files with the `cp` command works similarly to how you would expect on a Windows or Mac machine. The context around the move or copy operation determines what the result will be. For example, we could move and/or copy files into folders:

```
$ mv file_a folder1/
$ mv file_b folder2/
$ cp file_c folder3/
```

Before listing the results with `ls`, try to guess what the result will be.

```
$ ls
file_c folder1  folder2  folder3
$ ls folder1
file_a
$ ls folder2
file_b
$ ls folder3
file_c
```

Two files have been moved into folders, and `file_c` has been copied - so there is still a copy of `file_c` in the home directory. Move and copy commands can also be used to change the name of a file:

```
$ cp file_c file_c_copy
$ mv file_c file_c_new_name
```

By now, you may have found that Linux is very unforgiving with typos. Generous use of the `<Tab>` key to auto-complete file and folder names, as well as the `<UpArrow>` to cycle back through command history, will greatly improve the experience. As a general rule, try not to use spaces or strange characters in files or folder names. Stick to:

```
A-Z     # capital letters
a-z     # lowercase letters
0-9     # digits
-       # hyphen
_       # underscore
.       # period
```

Before we move on, let's clean up once again by removing the files and folders we have created. Do you remember the command for removing non-empty folders?

```
$ rm -r folder1
$ rm -r folder2
$ rm -r folder3
```

How do we remove `file_c_copy` and `file_c_new_name`?

```
$ rm file_c_copy
$ rm file_c_new_name
```

3.4 Looking at the Contents of Files

Everything we have seen so far has been with empty files and folders. We will now start looking at some real data. Navigate to your home directory, then issue the following `cp` command to copy a public file on the server to your local space:

```
$ cd ~     # the tilde ~ is also a shortcut referring to your home directory
$ pwd
/home/03439/wallen
$ cp /usr/share/dict/words .
$ ls
words
```

Try to use `<Tab>` to autocomplete the name of the file. Also, please notice the single dot `.` at the end of the copy command, which indicates that you want to cp the file to `.`, this present location (your home directory).

This `words` file is a standard file that can be found on most Linux operating systems. It contains 479,828 words, each word on its own line. To see the contents of a file, use the `cat` command to print it to screen:

```
$ cat words
1080
10-point
10th
11-point
12-point
16-point
18-point
1st
2
20-point
```

This is a long file! Printing everything to screen is much too fast and not very useful. We can use a few other commands to look at the contents of the file with `more` control:

```
$ more words
```

Press the `<Enter>` key to scroll through line-by-line, or the `<Space>` key to scroll through page-by-page. Press `q` to quit the view, or `<Ctrl+c>` to force a quit if things freeze up. A `%` indicator at the bottom of the screen shows your progress through the file. This is still a little bit messy and fills up the screen. The `less` command has the same effect, but is a little bit cleaner:

```
$ less words
```

Scrolling through the data is the same, but now we can also search the data. Press the `/` forward slash key, and type a word that you would like to search for. The screen will jump down to the first match of that word. The `n` key will cycle through other matches, if they exist.

Finally, you can view just the beginning or the end of a file with the `head` and `tail` commands. For example:

```
$ head words
$ tail words
```

The > and >> shortcuts in Linux indicate that you would like to redirect the output of one of the commands above. Instead of printing to screen, the output can be redirected into a file:

```
$ cat words > words_new.txt
$ head words > first_10_lines.txt
```

A single greater than sign > will redirect and **overwrite** any contents in the target file. A double greater than sign >> will redirect and **append** any output to the end of the target file.

One final useful way to look at the contents of files is with the `grep` command. `grep` searches a file for a specific pattern, and returns all lines that match the pattern. For example:

```
$ grep "banana" words
banana
bananaquit
bananas
cassabanana
```

Although it is not always necessary, it is safe to put the search term in quotes.

3.5 Network and File Transfers

In order to login or transfer files to a remote Linux file system, you must know the hostname (unique network identifier) and the username. If you are already on a Linux file system, those are easy to determine using the following commands:

```
$ whoami
wallen
$ hostname -f
login1.longhorn.tacc.utexas.edu
```

Given that information, a user would remotely login to this Linux machine using ssh in a Terminal:

```
[local]$ ssh wallen@longhorn.tacc.utexas.edu
enter password
enter 6-digit token
[longhorn]$
```

Windows users would typically use the program **PuTTY** (or another SSH client) to perform this operation. Logging out of a remote system is done using the `logout` command, or the shortcut `<Ctrl+d>`:

```
[longhorn]$ logout
[local]$
```

Copying files from your local computer to your home folder on Longhorn would require the `scp` command (Windows users use a client "WinSCP"):

```
[local]$ scp my_file wallen@longhorn.tacc.utexas.edu:/home/03439/wallen/
enter password
enter 6-digit token
```

In this command, you specify the name of the file you want to transfer (`my_file`), the username (`wallen`), the hostname (`longhorn.tacc.utexas.edu`), and the path you want to put the file (`/home/03439/wallen/`). Take careful notice of the separators including spaces, the `@` symbol, and the `:`.

Copy files from Longhorn to your local computer using the following:

```
[local]$ scp wallen@longhorn.tacc.utexas.edu:/home/03439/wallen/my_file ./
enter password
enter 6-digit token
```

Instead of files, full directories can be copied using the "recursive" flag (`scp -r ...`). The `rsync` tool is an advanced copy tool that is useful for synching data between two sites. Although we will not go into depth here, example `rsync` usage is as follows:

```
$ rsync -azv local remote
$ rsync -azv remote local
```

This is just the basics of copying files. See example scp usage and example rsync usage for more info.

3.6 Text Editing with VIM

VIM is a text editor used on Linux file systems.

Open a file (or create a new file if it does not exist):

```
$ vim file_name
```

There are two "modes" in VIM that we will talk about today. They are called "insert mode" and "normal mode". In insert mode, the user is typing text into a file as seen through the terminal (think about typing text into TextEdit or Notepad). In normal mode, the user can perform other functions like save, quit, cut and paste, find and replace, etc. (think about clicking the menu options in TextEdit or Notepad). The two main keys to remember to toggle between the modes are `i` and `Esc`.

Entering VIM insert mode:

```
> i
```

Entering VIM normal mode:

```
> Esc
```

A summary of the most important keys to know for normal mode are:

```
# Navigating the file:

arrow keys        move up, down, left, right
    Ctrl+u        page up
    Ctrl+d        page down

         0        move to beginning of line
         $        move to end of line

        gg        move to beginning of file
         G        move to end of file
        :N        move to line N

# Saving and quitting:

        :q        quit editing the file
        :q!       quit editing the file without saving

        :w        save the file, continue editing
        :wq       save and quit
```

3.7 Review of Topics Covered

Part 1: Creating and navigating folders

Command	Effect
`pwd`	print working directory
`ls`	list files and directories
`ls -l`	list files in column format
`mkdir dir_name/`	make a new directory
`cd dir_name/`	navigate into a directory
`rmdir dir_name/`	remove an empty directory
`rm -r dir_name/`	remove a directory and its contents
`.` or `./`	refers to the present location
`..` or `../`	refers to the parent directory

Part 2: Creating and manipulating files

Command	Effect
`touch file_name`	create a new file
`rm file_name`	remove a file
`rm -r dir_name/`	remove a directory and its contents
`mv file_name dir_name/`	move a file into a directory
`mv old_file new_file`	change the name of a file
`mv old_dir/ new_dir/`	change the name of a directory
`cp old_file new_file`	copy a file
`cp -r old_dir/ new_dir/`	copy a directory
`<Tab>`	autocomplete file or folder names
`<UpArrow>`	cycle through command history

Part 3: Looking at the contents of files

Command	Effect
`cat file_name`	print file contents to screen
`cat file_name >> new_file`	redirect output to new file
`more file_name`	scroll through file contents
`less file_name`	scroll through file contents
`head file_name`	output beginning of file
`tail file_name`	output end of a file
`grep pattern file_name`	search for ‘pattern’ in a file
`~/`	shortcut for home directory
`<Ctrl+c>`	force interrupt
`>`	redirect and overwrite
`>>`	redirect and append

Part 4: Network and file transfers

Command	Effect
`hostname -f`	print hostname
`whoami`	print username
`ssh username@hostname`	remote login
`logout`	logout
`scp local remote`	copy a file from local to remote
`scp remote local`	copy a file from remote to local
`rsync -azv local remote`	sync files between local and remote
`rsync -azv remote local`	sync files between remote and local
`<Ctrl+d>`	logout of host

Part 5: Text editing with VIM

Command	Effect
`vim file.txt`	open "file.txt" and edit with `vim`
`i`	toggle to insert mode
`<Esc>`	toggle to normal mode
`<arrow keys>`	navigate the file
`:q`	quit ending the file
`:q!`	quit editing the file without saving
`:w`	save the file, continue editing
`:wq`	save and quit

3.8 Additional Resources

- Practice Linux commands safely in a web-based emulator
- This is a good summary of the important commands you need to know
- Practice VIM in a web browser
- Practice VIM on the command line by typing `vimtutor`

CHAPTER

FOUR

PYTHON ESSENTIALS

After working through this material, attendees should be able to:

- Write and execute Python code on a remote server
- Use variables, lists, and dictionaries in Python
- Write conditionals using a variety of comparison operators
- Write useful while and for loops
- Arrange code into clean, well organized functions
- Read input from and write output to a file
- Import and use standard and non-standard Python libraries

Topics covered in this module include:

- Data types and variables (ints, floats, bools, strings, type(), print())
- Arithmetic operations (+, -, *, /, **, %, //)
- Lists and dictionaries (creating, interpreting, appending)
- Conditionals and control loops (comparison operators, if/elif/else, while, for, break, continue, pass)
- Functions (defining, passing arguments, returning values)
- File handling (open, with, read(), readline(), strip(), write())
- Importing libraries (import, random, names, pip)

4.1 Log in to Longhorn

To log in to `longhorn.tacc.utexas.edu`, follow the instructions for your operating system or ssh client below.

Mac / Linux

```
Open the application 'Terminal'
ssh username@longhorn.tacc.utexas.edu
(enter password)
(enter 6-digit token)
```

Windows

```
Open the application 'PuTTY'
enter Host Name: longhorn.tacc.utexas.edu
(click 'Open')
(enter username)
(enter password)
(enter 6-digit token)
```

If you can't access Longhorn yet, a local or web-based Python 3 environment will work for this guide. However, you will need to access Longhorn for future materials.

Try this Python 3 environment in a browser.

Note: For the first few sections below, we will be using the Python interpreter in *interactive mode* to try out different things. Later on when we get to more complex code, we will be saving the code in files (scripts) and invoking the interpreter non-interactively.

4.2 Data Types and Variables

Start up the interactive Python interpreter:

```
[longhorn]$ python3
Python 3.6.8 (default, Apr 25 2019, 20:47:23)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Tip: To exit the interpreter, type `quit()`.

The most common data types in Python are similar to other programming languages. For this workshop, we probably only need to worry about **integers**, **floats**, **booleans**, and **strings**.

Assign some values to variables by doing the following:

```
>>> my_int = 5
>>> my_float = 5.0
>>> my_bool = True      # or False, notice capital letters
>>> my_string = 'Hello, world!'
```

In Python, you don't have to declare type. Python figures out the type automatically. Check using the `type()` function:

```
>>> type(my_int)
<class 'int'>
>>> type(my_float)
<class 'float'>
>>> type(my_bool)
<class 'bool'>
>>> type(my_string)
<class 'str'>
```

Print the values of each variable using the `print()` function:

```
>> print(my_int)
5
>> print('my_int')
my_int
```

(Try printing the others as well). And, notice what happens when we print with and without single quotes? What is the difference between `my_int` and `'my_int'`?

You can convert between types using a few different functions. For example, when you read in data from a file, numbers are often read as strings. Thus, you may want to convert the string to integer or float as appropriate:

```
>>> str(my_int)       # convert int to string
>>> str(my_float)     # convert float to string
>>> int(my_string)    # convert string to int
>>> float(my_string)  # convert string to float
>>>
>>> value = 5
>>> print(value)
5
>>> type(value)
<class 'int'>
>>> new_value = str(value)
>>> print(new_value)
'5'
>>> type(new_value)
<class 'str'>
```

4.3 Arithmetic Operations

Next, we will look at some basic arithmetic. You are probably familiar with the standard operations from other languages:

```
Operator   Function          Example   Result
+          Addition          1+1       2
-          Subtraction       9-5       4
*          Multiplication    2*2       4
/          Division          8/4       2
**         Exponentiation    3**2      9
%          Modulus           5%2       1
//         Floor division    5//2      2
```

Try a few things to see how they work:

```
>>> print(2+2)
>>> print(355/113)
>>> print(10%9)
>>> print(3+5*2)
>>> print('hello' + 'world')
>>> print('some' + 1)
>>> print('number' * 5)
```

Also, carefully consider how arithmetic options may affect type:

```
>>> number1 = 5.0/2
>>> type(number1)
```

(continues on next page)

(continued from previous page)

```
<class 'float'>
>>> print(number1)
2.5
>>> number2 = 5/2
>>> type(number2)
<class 'float'>
>>> print(number2)
2.5
>>> print(int(number2))
2
```

4.4 Lists and Dictionaries

Lists are a data structure in Python that can contain multiple elements. They are ordered, they can contain duplicate values, and they can be modified. Declare a list with square brackets as follows:

```
>>> my_shape_list = ['circle', 'triangle', 'square', 'diamond']
>>> type(my_shape_list)
<class 'list'>
>>> print(my_shape_list)
['circle', 'triangle', 'square', 'diamond']
```

Access individual list elements:

```
>>> print(my_shape_list[0])
circle
>>> type(my_shape_list[0])
<class 'str'>
>>> print(my_shape_list[2])
square
```

Create an empty list and add things to it:

```
>>> my_number_list = []
>>> my_number_list.append(5)      # 'append()' is a method of the list class
>>> my_number_list.append(6)
>>> my_number_list.append(2)
>>> my_number_list.append(2**2)
>>> print(my_number_list)
[5, 6, 2, 4]
>>> type(my_number_list)
<class 'list'>
>>> type(my_number_list[1])
<class 'int'>
```

Lists are not restricted to containing one data type. Combine the lists together to demonstrate:

```
>>> my_big_list = my_shape_list + my_number_list
>>> print(my_big_list)
['circle', 'triangle', 'square', 'diamond', 5, 6, 2, 4]
```

Another way to access the contents of lists is by slicing. Slicing supports a start index, stop index, and step taking the form: `mylist[start:stop:step]`. Only the first colon is required. If you omit the start, stop, or :step, it is assumed you mean the beginning, end, and a step of 1, respectively. Here are some examples of slicing:

```
>>> mylist = ['thing1', 'thing2', 'thing3', 'thing4', 'thing5']
>>> print(mylist[0:2])      # returns the first two things
['thing1', 'thing2']
>>> print(mylist[:2])       # if you omit the start index, it assumes the beginning
['thing1', 'thing2']
>>> print(mylist[-2:])      # returns the last two things (omit the stop index and it␣
↪assumes the end)
['thing4', 'thing5']
>>> print(mylist[:])        # returns the entire list
['thing1', 'thing2', 'thing3', 'thing4', 'thing5']
>>> print(mylist[::2])      # return every other thing (step = 2)
['thing1', 'thing3', 'thing5']
```

Note: If you slice from a list, it returns an object of type list. If you access a list element by its index, it returns an object of whatever type that element is. The choice of whether to slice from a list, or iterate over a list by index, will depend on what you want to do with the data.

Dictionaries are another data structure in Python that contain key:value pairs. They are always unordered, they cannot contain duplicate keys, and they can be modified. Create a new dictionary using curly brackets:

```
>>> my_shape_dict = {
...   'most_favorite': 'square',
...   'least_favorite': 'circle',
...   'pointiest': 'triangle',
...   'roundest': 'circle'
... }
>>> type(my_shape_dict)
<class 'dict'>
>>> print(my_shape_dict)
{'most_favorite': 'square', 'least_favorite': 'circle', 'pointiest': 'triangle',
↪'roundest': 'circle'}
>>> print(my_shape_dict['most_favorite'])
square
```

As your preferences change over time, so to can values stored in dictionaries:

```
>>> my_shape_dict['most_favorite'] = 'rectangle'
>>> print(my_shape_dict['most_favorite'])
rectangle
```

Add new key:value pairs to the dictionary as follows:

```
>>> my_shape_dict['funniest'] = 'squircle'
>>> print(my_shape_dict['funniest'])
squircle
```

Many other methods exist to access, manipulate, interpolate, copy, etc., lists and dictionaries. We will learn more about them out as we encounter them later in this course.

4.5 Conditionals and Control Loops

Python **comparison operators** allow you to add conditions into your code in the form of `if` / `elif` / `else` statements. Valid comparison operators include:

Operator	Comparison	Example	Result
==	Equal	1==2	False
!=	Not equal	1!=2	True
>	Greater than	1>2	False
<	Less than	1<2	True
>=	Greater than or equal to	1>=2	False
<=	Less Than or equal to	1<=2	True

A valid conditional statement might look like:

```
>>> num1 = 10
>>> num2 = 20
>>>
>>> if (num1 > num2):                    # notice the colon
...     print('num1 is larger')          # notice the indent
... elif (num2 > num1):
...     print('num2 is larger')
... else:
...     print('num1 and num2 are equal')
```

In addition, conditional statements can be combined with **logical operators**. Valid logical operators include:

Operator	Description	Example
and	Returns True if both are True	a < b and c < d
or	Returns True if at least one is True	a < b or c < d
not	Negate the result	not(a < b)

For example, consider the following code:

```
>>> num1 = 10
>>> num2 = 20
>>>
>>> if (num1 < 100 and num2 < 100):
...     print('both are less than 100')
... else:
...     print('at least one of them is not less than 100')
```

While loops also execute according to conditionals. They will continue to execute as long as a condition is True. For example:

```
>>> i = 0
>>>
>>> while (i < 10):
...     print( f'i = {i}' )        # literal string interpolation
...     i = i + 1
```

The `break` statement can also be used to escape loops:

```
>>> i = 0
>>>
>>> while (i < 10):
...     print( f'i = {i}' )
```

(continues on next page)

(continued from previous page)

```
...     i = i + 1
...     if (i==5):
...         break
...     else:
...         continue
```

For loops in Python are useful when you need to execute the same set of instructions over and over again. They are especially great for iterating over lists:

```
>>> my_shape_list = ['circle', 'triangle', 'square', 'diamond']
>>>
>>> for shape in my_shape_list:
...     print(shape)
>>>
>>> for shape in my_shape_list:
...     if (shape == 'circle'):
...         pass                    # do nothing
...     else:
...         print(shape)
```

You can also use the `range()` function to iterate over a range of numbers:

```
>>> for x in range(10):
...     print(x)
>>>
>>> for x in range(10, 100, 5):
...     print(x)
>>>
>>> for a in range(3):
...     for b in range(3):
...         for c in range(3):
...             print( f'{a} + {b} + {c} = {a+b+c}' )
```

Note: The code is getting a little bit more complicated now. It will be better to stop running in the interpreter's interactive mode, and start writing our code in Python scripts.

4.6 Functions

Functions are blocks of codes that are run only when we call them. We can pass data into functions, and have functions return data to us. Functions are absolutely essential to keeping code clean and organized.

On the command line, use a text editor to start writing a Python script:

```
[longhorn]$ vim function_test.py
```

Enter the following text into the script:

```
1 def hello_world():
2     print('Hello, world!')
3
4 hello_world()
```

After saving and quiting the file, execute the script (Python code is not compiled - just run the raw script with the `python3` executable):

```
[longhorn]$ python3 function_test.py
Hello, world!
```

Note: Future examples from this point on will assume familiarity with using the text editor and executing the script. We will just be showing the contents of the script and console output.

More advanced functions can take parameters and return results:

```
def add5(value):
    return(value + 5)

final_number = add5(10)
print(final_number)
```

```
15
```

Pass multiple parameters to a function:

```
def add5_after_multiplying(value1, value2):
    return( (value1 * value2) + 5)

final_number = add5_after_multiplying(10, 2)
print(final_number)
```

```
25
```

It is a good idea to put your list operations into a function in case you plan to iterate over multiple lists:

```
def print_ts(mylist):
    for x in mylist:
        if (x[0] == 't'):      # a string (x) can be interpreted as a list of chars!
            print(x)

list1 = ['circle', 'triangle', 'square', 'diamond']
list2 = ['one', 'two', 'three', 'four']

print_ts(list1)
print_ts(list2)
```

```
triangle
two
three
```

There are many more ways to call functions, including handing an arbitrary number of arguments, passing keyword / unordered arguments, assigning default values to arguments, and more.

4.7 File Handling

The `open()` function does all of the file handling in Python. It takes two arguments - the *filename* and the *mode*. The possible modes are read (`r`), write (`w`), append (`a`), or create (`x`).

For example, to read a file do the following:

```
with open('/usr/share/dict/words', 'r') as f:
    for x in range(5):
        print(f.readline())
```

```
1080

10-point

10th

11-point

12-point
```

Tip: By opening the file with the `with` statement above, you get built in exception handling, and it automatically will close the file handle for you. It is generally recommended as the best practice for file handling.

You may have noticed in the above that there seems to be an extra space between each word. What is actually happening is that the file being read has newline characters on the end of each line (`\n`). When read into the Python script, the original new line is being printed, followed by another newline added by the `print()` function. Stripping the newline character from the original string is the easiest way to solve this problem:

```
with open('/usr/share/dict/words', 'r') as f:
    for x in range(5):
        print(f.readline().strip('\n'))
```

```
1080
10-point
10th
11-point
12-point
```

Read the whole file and store it as a list:

```
words = []

with open('/usr/share/dict/words', 'r') as f:
    words = f.read().splitlines()                # careful of memory usage

for x in range(5):
    print(words[x])
```

```
1080
10-point
10th
11-point
12-point
```

Write output to a new file on the file system; make sure you are attempting to write somwhere where you have permissions to write:

```
my_shapes = ['circle', 'triangle', 'square', 'diamond']

with open('my_shapes.txt', 'w') as f:
    for shape in my_shapes:
        f.write(shape)
```

```
(in my_shapes.txt)
circletrianglesquarediamond
```

You may notice the output file is lacking in newlines this time. Try adding newline characters to your output:

```
my_shapes = ['circle', 'triangle', 'square', 'diamond']

with open('my_shapes.txt', 'w') as f:
    for shape in my_shapes:
        f.write( f'{shape}\n' )
```

```
(in my_shapes.txt)
circle
triangle
square
diamond
```

Now notice that the original line in the output file is gone - it has been overwritten. Be careful if you are using write (`w`) vs. append (`a`).

4.8 Importing Libraries

The Python built-in functions, some of which we have seen above, are useful but limited. Part of what makes Python so powerful is the huge number and variety of libraries that can be *imported*. For example, if you want to work with random numbers, you have to import the 'random' library into your code, which has a method for generating random numbers called 'random'.

```
import random

for i in range(5):
    print(random.random())
```

```
0.47115888799541383
0.5202615354150987
0.8892412583071456
0.7467080997595558
0.025668541754695906
```

More information about using the `random` library can be found in the Python docs

Some libraries that you might want to use are not included in the official Python distribution - called the *Python Standard Library*. Libraries written by the user community can often be found on PyPI.org and downloaded to your local environment using a tool called `pip3`.

For example, if you wanted to download the names library and use it in your Python code, you would do the following:

```
[longhorn]$ pip3 install --user names
Collecting names
  Downloading https://files.pythonhosted.org/packages/44/4e/
↪f9cb7ef2df0250f4ba3334fbdabaa94f9c88097089763d8e85ada8092f84/names-0.3.0.tar.gz␣
↪(789kB)
    100% || 798kB 1.1MB/s
Installing collected packages: names
  Running setup.py install for names ... done
Successfully installed names-0.3.0
```

Notice the library is installed above with the `--user` flag. Longhorn is a shared system and non-privileged users can not download or install packages in root locations. The `--user` flag instructs `pip3` to install the library in your own home directory.

```
1 import names
2
3 for i in range(5):
4     print(names.get_full_name())
```

```
Johnny Campbell
Lawrence Webb
Johnathan Holmes
Mary Wang
Jonathan Henry
```

4.9 Exercises

Test your understanding of the materials above by attempting the following exercises.

- Create a list of ~10 different integers. Write a function (using modulus and conditionals) to determine if each integer is even or odd. Print to screen each digit followed by the word 'even' or 'odd' as appropriate.
- Using nested for loops and if statements, write a program that iterates over every integer from 3 to 100 (inclusive) and prints out the number only if it is a prime number.
- Create three lists containing 10 integers each. The first list should contain all the integers sequentially from 1 to 10 (inclusive). The second list should contain the squares of each element in the first list. The third list should contain the cubes of each element in the first list. Print all three lists side-by-side in three columns. E.g. the first row should contain 1, 1, 1 and the second row should contain 2, 4, 8.
- Write a script to read in /usr/share/dict/words and print just the last 10 lines of the file. Write another script to only print words beginning with the letters "pyt".

4.10 Additional Resources

- The Python Standard Library
- PEP 8 Python Style Guide
- Python3 environment in a browser
- Jupyter Notebooks in a browser

CHAPTER

FIVE

NUMPY, SCIPY, AND MATPLOTLIB PRIMER

NumPy, SciPy, and Matplotlib are three complementary and important Python libraries that are useful for exploratory data analysis and machine learning. This primer is a high-level introduction to each library, providing short examples for each.

5.1 NumPy

NumPy provides a multidimensional array object for very fast operations on numerical data (e.g. arithmetic, logical, sorting, selecting, transforming, statistical, etc.).

Much faster than doing similar operations with normal Python data types because it uses pre-complied, vectorized C code behind the scenes.

```
import numpy as np

a = np.arange(15).reshape(3, 5)

print(a)
print(a.shape)
print(a.ndim)
```

```
### Output:

[[ 0,  1,  2,  3,  4],
 [ 5,  6,  7,  8,  9],
 [10, 11, 12, 13, 14]]
(3, 5)
2
```

A few very easy array operation examples:

```
import numpy as np

a = np.arange(15).reshape(3, 5)

print(a.max())
print(a.min())
print(a.sum())
print(a*2)
```

```
### Output:

```

(continues on next page)

(continued from previous page)

```
14
0
105
[[ 0  2  4  6  8]
 [10 12 14 16 18]
 [20 22 24 26 28]]
```

5.2 SciPy

SciPy provides mathematical algorithms and convenience functions for working with NumPy arrays.

It comes with a host of high level commands for different scientific computing domains, including clustering, Fourier transforms, integration, interpolation, linear algebra, signal processing, among others.

The following short code block defines a function to be integrated (called the "integrand") as `f(x) = ax^2 + b`. The SciPy `quad` function evaluates the integral over a given range, in this case from 0 to 1. The output is a tuple where the first item is the value of the integral, and the second item is the estimated error.

```
from scipy.integrate import quad
def integrand(x, a, b):
    return a*x**2 + b

a = 2
b = 1
I = quad(integrand, 0, 1, args=(a,b))
print(I)
```

```
### Output:

(1.6666666666666667, 1.8503717077085944e-14)
```

5.3 Matplotlib

Matplotlib provides utilities for creating static, animated, and interactive visualizations of data.

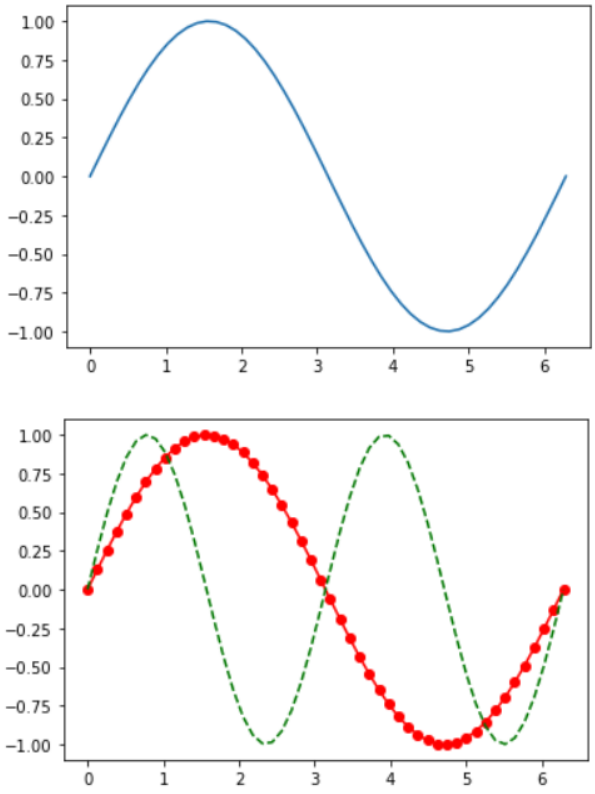
You can use it to create many different types of plots (line, histogram, scatter, countour, 3D, etc.) with full control over all labels, colors, styles, etc. Look online for examples of the kind of plot you want to make, and undoubtedly there will be some Matplotlib samples available.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 50)
plt.plot(x, np.sin(x))
plt.show()
```

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 50)
plt.plot(x, np.sin(x), 'r-o', x, np.sin(2*x), 'g--')
plt.show()
```




```
# save the image to file instead
plt.savefig('my_sinwave.png')
```

5.4 Additional Resources

- NumPy Docs
- SciPy Docs
- Matplotlib Docs

CHAPTER

SIX

ADDITIONAL RESOURCES

- Python for Machine Learning: https://learn.tacc.utexas.edu/mod/page/view.php?id=62
- AI for Everyone: https://www.coursera.org/learn/ai-for-everyone/home/welcome