Intro to Unix

Galvanize Data Science Immersive

Week 1, Day 1, Lecture 1

Land Belenky

galvanize

galvanize

Outline

- Define an Operating System
- Brief History of Unix & Linux
- Linux File Structure
- Shells and Scripts
- Directory Commands
- File Commands
- → **Python Programming** ←

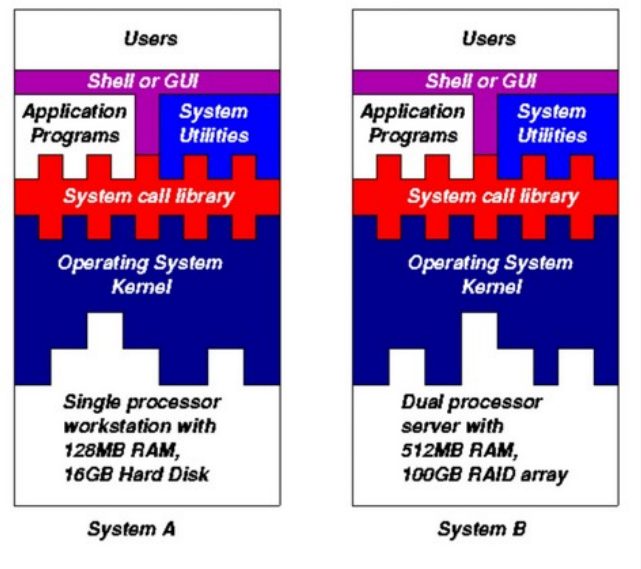
galvanize

Objective:

After this lecture, you should be able to:

- Create and navigate directories
- Create and navigate files
- Get started in python in at least two modes

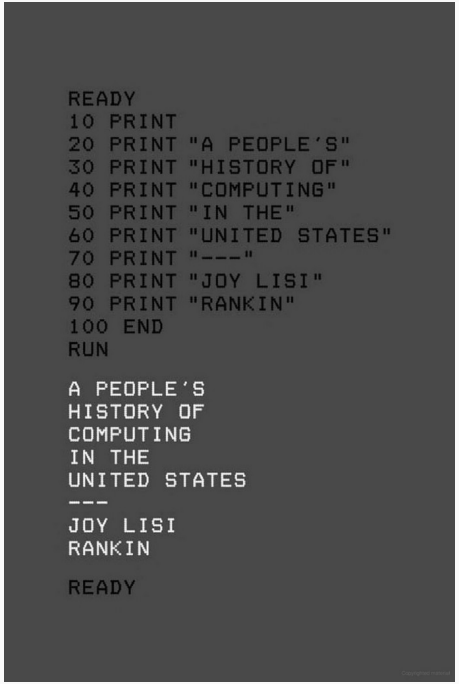
Operating Systems



- A set of software routines that allow users and applications to access system resources (CPU, memory, network, storage, etc)
- Saves us the trouble of having to know exactly how the computer is built and configured:
  - provides a uniform interface for all different types and sizes of computers
- Protects the computer from accidental (or deliberate) mismanagement of resources

Brief History of Unix

- Developed at Bell Labs 1969~1971
- Designed for multi-user, multi-tasking, time-sharing, networking
- One large computer, multiple connected “terminals” each working in turn
- TTY Terminals:
  - text-only entry (no GUI)
  - line input → line output
  - No scrolling up, no editing previous lines
- Reputation for being cryptic and obscure
- Largely supplanted by Windows circa 1992 for business applications, thanks in large part to friendly GUI

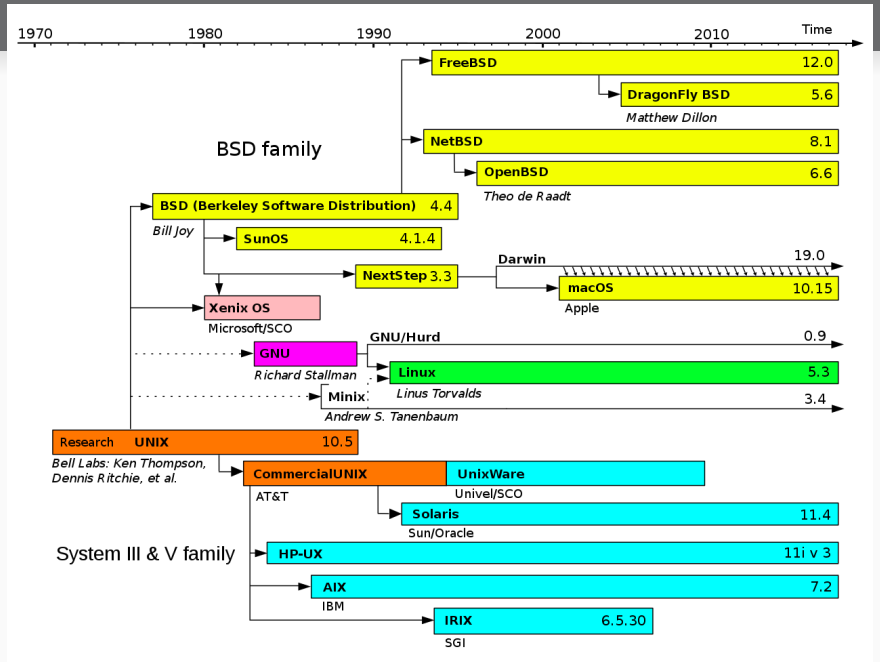


Linux

- Linus Torvalds: (b. Helsinki, Finland, 1969)
  - Started ~1990
  - Free, open-source, community supported
- Secure and Reliable
- Hundreds of variants
- Majority of internet servers
- 100% of the TOP500 supercomputers
- Extremely customizable
- Massive (and helpful) user knowledge base

MacOS

- Derived from UNIX
- Based on BSD/NextStep/Darwin
- A cousin to Linux
- Shares many common commands and structure
- Not completely compatible
  - Linux programs may or may not run on Mac and vice-versa



Kernel, Shell and Windows

- “**kernel**”:
  - most basic and fundamental part of operating system (I.e, the real OS)
  - handles machine-level processes (CPU, Memory, Storage, Input-Output devices)
  - We do not interact directly with the kernel
- “**Shell**”:
  - frequently called “**terminal window**” or “**command line**”
    - Some will argue that this is not strictly correct
  - Our primary way of interacting with the computer.
  - Sends commands to kernel
- “**Window Manager**”
  - program to enable drawing of windows
- “**Desktop Environment**”
  - set of programs to relay commands to kernel
  - a graphical analog of shell

Virtual Terminals in MacOS and Linux

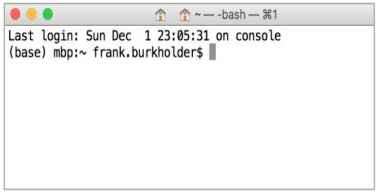
Modeled on old-fashioned tty

Line entry/line output

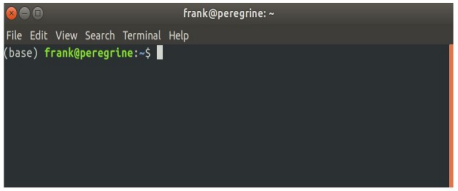
No editing of previous input

Different options for shell:

- bash (Ubuntu and old Mac default)
- zsh (New Mac default)



MacOs:
command - space
(to open Spotlight)
then type
“terminal”



Ubuntu Linux:
ctrl - alt - t

Advantages of the Command-Line Interface

- Text entry (from keyboard) faster and more accurate than mouse
  - Takes some practice
- Text entry more customizable
  - Use aliases for commonly used commands, rather than repeating sequences of mouse movements
- Text-based commands can be saved in a file, or generated from a file. File can be stored, distributed, re-run and modified.
  - Document your processes and distribute them to other computers or users
- Programs can be run by other programs
  - Automated or semi-automated management of clusters: Docker, Kubernetes, DevOPs

galvanize

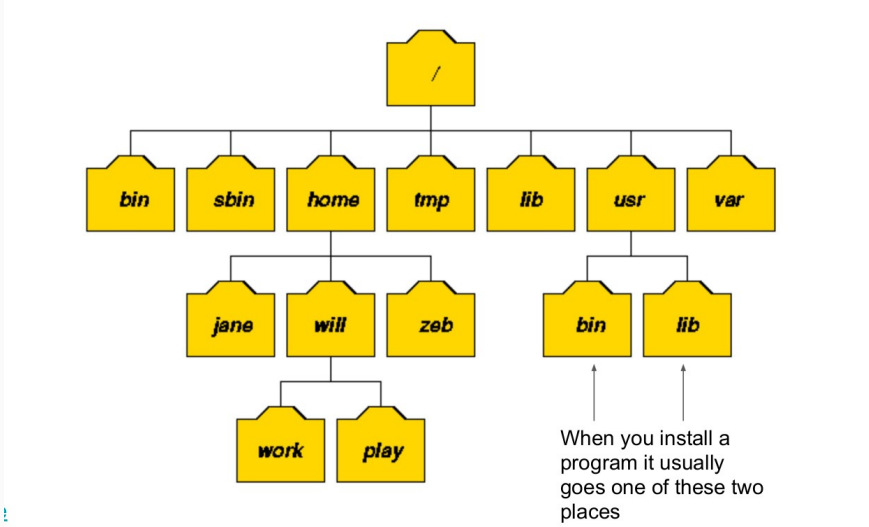
UNIX/Linux Design Philosophy

Mike Gancarz, The Unix Philosophy, 1994

- Small is beautiful
- Make each program do one thing well
- Build a prototype as soon as possible
- Choose portability over efficiency
- Store data in flat text files
- Use software leverage to your advantage
- Use shell scripts to increase leverage and portability
- Avoid captive user interfaces
- Make every program a filter

Unix Organization

- “Everything is a file”
- Frequently a *readable* text file
- All files are in a directory
- All directories (except root) in a directory
- Root directory: “**/**”
- Tree structure



galvanize

Directory Navigation

- New Terminal: Home Directory (~)
- **pwd**: “Print working directory”
- **$PWD:** variable of working directory
- **echo $HOME** “print home directory”
- **cd**: “Change directory”
  - **cd** → Go home (from anywhere)
  - **cd ..** → go up one level
  - **cd ../../..** → go up 3 levels

- **cd <directory_name>** → go to sub-directory of this folder (relative path)
- **cd /home/user/Documents** → absolute path
- **cd ../parent/child** → relative path
- **mkdir <new_directory>** → make new directory
- **rm -rf <directory>** → delete directory
  - Careful! May not be able to recover!

File and Directory Tips

- Linux uses forward-slash (/) in paths. (Windows uses backslash)
- Keep all your work in a folder in your home directory:
  - **`~/Galvanize/assignments`**
  - **`~/Galvanize/capstones`**
- You can use spaces, but it is better not to.
  - You will need to use quotation marks or backslashes to escape spaces
  - Use_underscore_character_for_spaces
- Capitalization matters!
- Use <tab> auto-complete
  - Type just enough letters to resolve the name, then hit tab.

Contents of a Directory

- **`ls`** → “list contents” (non-hidden files and directories)
- Files or directories beginning with “.” are ‘hidden’
- **`ls /child_directory`**
- **`ls ..`**
- **`ls ~/Documents`**
- **`ll` (alias for `ls -alF`)** → vertical list, including hidden, with indicator

- Command Line Options:
  - Use “man” to get see help menu
    - e.g.: **`man ls`**
  - single-letter options are preceded with a single dash
  - single-letter options can be grouped together
    - e.g.: **`ls -alF`**
  - word-length options preceded with double-dash, cannot be grouped
    - e.g.: **`ls --no-group --human-readable`**

File Ownership and Permission

- All files have an owner and an owner's group
- **`ls -l`** shows ownership and permission:

- **owner** = land
- **group** = land
- **`rwx`** = read, write, execute
  - "-" indicates permission denied
  - Owner, Group, Everyone Else

```
land@nitro unix (master) $ ls -l
total 9104
-rw-rw-r-- 1 land land 3645836 Jun 22 18:52 intro_unix.odg
-rw-rw-r-- 1 land land 4518732 Jun 22 18:52 just_enough_command_line.pdf
-rw-rw-r-- 1 land land   38289 Jun 22 18:52 Readme.md
-rw-rw-r-- 1 land land 1106848 Jun 24 14:32 unix.odp
```

Changing File Ownership and Permission

- **chown**: “change owner”
- **sudo**: “superuser do”
  - Warning! Can accidentally delete or modify important files!
- **chgrp**: “change group”

- **chmod**: “change file mode”
  - grant/deny:
    - read (4)
    - write (2)
    - execute (1)
  - e.g:
    - 6 = read and write
    - 7 = read, write and execute
    - 0 = no access
  - Repeat three times for user, group, everyone else

File Creation

Many different ways to create files:

- **`touch <filename>`**:
  - create new file, or update “last modified” on existing file
- **`echo Hello > hello.txt`**
  - create new file or over-write existing with contents “Hello”
- **`echo $PWD > pwd.txt`**
  - create new file or overwrite existing with working directory
- **`echo goodbye >> hello.txt`**
  - add ‘goodbye’ to end of existing file (or create new file if does not exist)
- **`history > history.txt`**
  - dump list of recent shell commands to file
- **`code .`**
  - open VS Code in the current directory
- **`code <filename>`**
  - open or create new file in VS Code

&, &&, |, ||, >>, and >

One of the most powerful features of shell commands is that they can be chained together

- **`<command 1> &`**: Run in background
  - Do not wait for command to complete before starting next command or returning control to terminal window
- **`<command 1> && <command 2>`**
  - run command 2 after command 1 finishes successfully
- **`<command 1> | <command 2>`**
  - Use the output of command 1 as the input to command 2
- **`<command 1> || <command 2>`**
  - Run command 2 if command 1 does not finish successfully

- **`command > file`**
  - send the output of command to a file, rather than to the monitor
  - Over-write contents of file
- **`command >> file`**
  - send output of command to end of file

Viewing Contents of a File

- **`cat <filename>`**
  - print contents of file to screen
- **`head <filename>`**
  - print first 10 lines of file to screen
- **`tail <filename>`**
  - print last 10 lines of file to screen
- **`more <filename>`**
  - print file one screen at a time
- **`less <filename>`**
  - same as more, but more options

May also re-direct output:

- **`head filename > new_file.txt`**
  - create new file with first 10 lines
- **`tail filename >> new_file.txt`**
  - Add last ten lines of file to end of new_file
- **`code <filename>`**
  - open file in VS Code for reading and/or editing

Move, Rename and Delete Files

- **`mv`**: move file <from> <to>
  - used to move and rename files or directories
  - may be within the same directory or across directories
- **`cp`**: copy file (or directory)
  - <existing> <new>
  - may be within or across directories

- **`rm`**: remove file
- **`rm -rf`**: remove directory
  - Note: the **`-rf`** options are to provide extra protection for deleting the contents of a directory

Editing Text Files

Generally, a full-featured text editor (VS Code) is preferred

- When VS code is not available, built-in text editors (VIM, nano, gedit) can be used
- Shell scripts can also be used to edit text files because nearly all shell commands are *filters*
  - <text in> → (modify/sort/filter/evaluate) → <text out>
- <Text in> can come from a file and <text out> can go to a file
- Therefore, there are many different ways to edit text files by stringing together these commands
- Some useful commands:
  - **`cat/head/tail/more/less`**
  - **`sed`**
  - **`grep`**
  - **`awk`**
  - **`perl`**

Environment Variables

- Values that need to be accessed frequently or programmatically can be stored by name as environment variables
- **`env`** to see a list of environment variables
- **`export`** used to set variable: (e.g. **`export MY_VALUE=12345`**)
- **`$`** used to recall values: (e.g. **`echo $MY_VALUE`** → 12345)
- Environment variables only persist for the duration of the terminal window in which they are defined
- Define environment variables in *profile* file so that they get loaded for every new terminal window
  - **`echo "export MY_VALUE=12345" >> ~/.bashrc`**
- Some useful environment variables:
  - **`$PATH`**: the list of directories where computer will look for programs to run
  - **`$HOME`**: your home directory
  - **`$PWD`**: current working directory
  - **`$SHELL`** or **`$0`**: the name and location of the shell program
  - **`$AWS_ACCESS_KEY`** and **`$AWS_SECRET_KEY`** (Keep these safe)

Aliases

- Aliases: shortcuts to run commands:
  - **`alias ll='ls -alF'`**
  - **`alias jup='jupyter notebook'`**
  - **`alias gh="git log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short"`**
  - **`alias goog="xdg-open https://google.com &"`**
- Use **`alias`** to see list of currently defined aliases
- Put alias commands in *profile* file to load at the start of every terminal session

galvanize

Profile file

- Used to set **`env`** variables, aliases, execute commands, etc
- Run automatically every time a terminal window is started
- Can also be run with **`source`** command (e.g. **`source ~/.bashrc`**)
- Depends on which shell you are running:
  - bash (Linux) → **`~/.bashrc`**
  - bash (Old Mac) → **`~/.bash_profile`**
  - zsh (New Mac) → **`~/.zshrc`**
- Use **`echo $0`** to determine which shell you are running
- can be edited with **`code ~/.bashrc`**
- can be appended with **`echo -e "\n new command" >> ~/.bashrc`**
  - Remember to append to file (**`>>`**), not over-write (**`>`**)!

galvanize

Python Programming Environments

VS Code + iPython (terminal)

- All you need is a text editor and a terminal window
- clean, text-only files (`.py`)
- Combination of script-based and interactive programming
- Easy to import into other scripts

Jupyter Notebook

- Rapid development with in-line output values, charts and markdown
- Python code intermixed with html
- difficult to import into other scripts
- Easy to accidentally execute out-of-order

galvanize

Python Programming Environments

In this class, you must be proficient with both **`.py`** scripts in the terminal window and **`jupyter notebooks`**.
Do not become overly reliant on one approach!

(A 3rd option: **`python filename.py`**)
We will use this approach less often and discuss later.

Hello World Examples

- ipython
- VS Code/ipython
- jupyter notebook
- python

Hello World in ipython

- Open a terminal window
- type “**ipython**”
- type

```
print('Hello World')
```

- See “**Hello World**”

- Advantages:
  - extremely fast and easy
  - all values retained in memory until `exit`
- Disadvantages:
  - code not saved in a file
  - editing in ipython is difficult
  - all values lost upon exit
- Suitable for:
  - back-of-envelope calculations
  - quick exploration of ideas

Hello World in VS Code/ipython

- Open a terminal window
- **`cd`** to directory
- **`code .`** to open VS Code
- type **`ipython`**
- In VS Code, create new file
- type **`print('Hello World')`**
- save as **`hello.py`**
- In ipython, type **`run hello.py`**

- Advantages:
  - all values retained in memory until `exit`
  - code saved to file
  - powerful editing tools in VS Code
  - **`.py`** files can be imported to other scripts
  - **`.py`** files can be executed programmatically
  - Simple text-only files easy to read
- Disadvantages:
  - No in-line charts or markdown formatting
- Suitable for:
  - All work
  - Ideal choice for complex projects

Hello World in Jupyter Notebook

- Open a terminal window
- **`jupyter notebook`** to launch in browser (or alias)
- Enter browser and create new python notebook
- type **`print('Hello World')`** in first cell
- **`<ctrl> -Enter`** to execute cell

- Advantages:
  - all values retained in memory until `exit`
  - code saved to file
  - In-line graphs and markdown comments
- Disadvantages:
  - .ipynb files difficult to import or execute
  - Potential for out-of-order execution
  - editing across cells can be difficult
  - html tags make files sloppy hard to diff
- Suitable for:
  - Proof of concept
  - Development of ideas
  - Presentations

galvanize

Hello World in python (command line)

- Open a terminal window
- navigate to directory
- **`code .`** to launch VS Code
- create new file,
- type **`print('Hello World')`**
- save as **`hello.py`**
- on command line, type:
  - **`python hello.py`**

- Advantages:
  - code saved to file
  - programmatic execution
  - dovetails with Ipython/VS Code approach
  - **`.py`** files can be imported to other scripts
- Disadvantages:
  - values not retained in memory upon completion
  - Always have to run full script, beginning to end
- Suitable for:
  - Finished projects
  - Automated execution

galvanize

Key Commands you **will** need

```
pwd
mkdir
cd
ls
cp
mv
rm
code .
code ~/.bashrc
ipython
jupyter notebook
```

galvanize

Final Thoughts

- The terminal window (shell) is your first stop for any task
- Avoid using mouse
- Use keyboard shortcuts:
  - **`<tab>`**: auto-complete
  - **up-arrow**: repeat recent command
  - **`<ctrl> -r`**: search recent commands
  - **`history | grep <command>`** to review recent commands
  - **`!<history number>`** to repeat a command by number
- **Must be proficient in both ipython and jupyter notebook**
- Use **`man`** to learn about shell commands
- Use VS Code (when available) for general text editing