# Introduction to the Command Line and Bash

Brent McPherson

2024-05-13

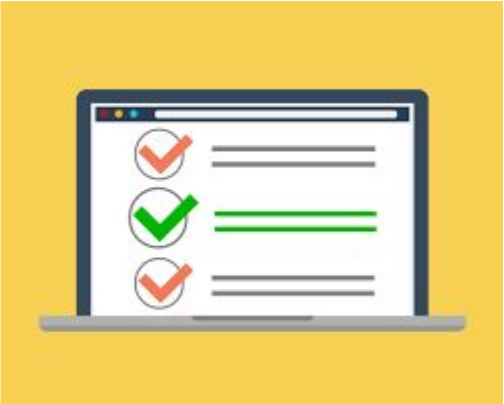
ORIGAMI Lab

neuro
Montreal Neurological Institute-Hospital

McGill University

# Before we get started

- Course materials are here: https://github.com/neurodatascience/QLS-course-materials/tree/main/Lectures/2024
- We'll use materials from here as part of the exercises later.



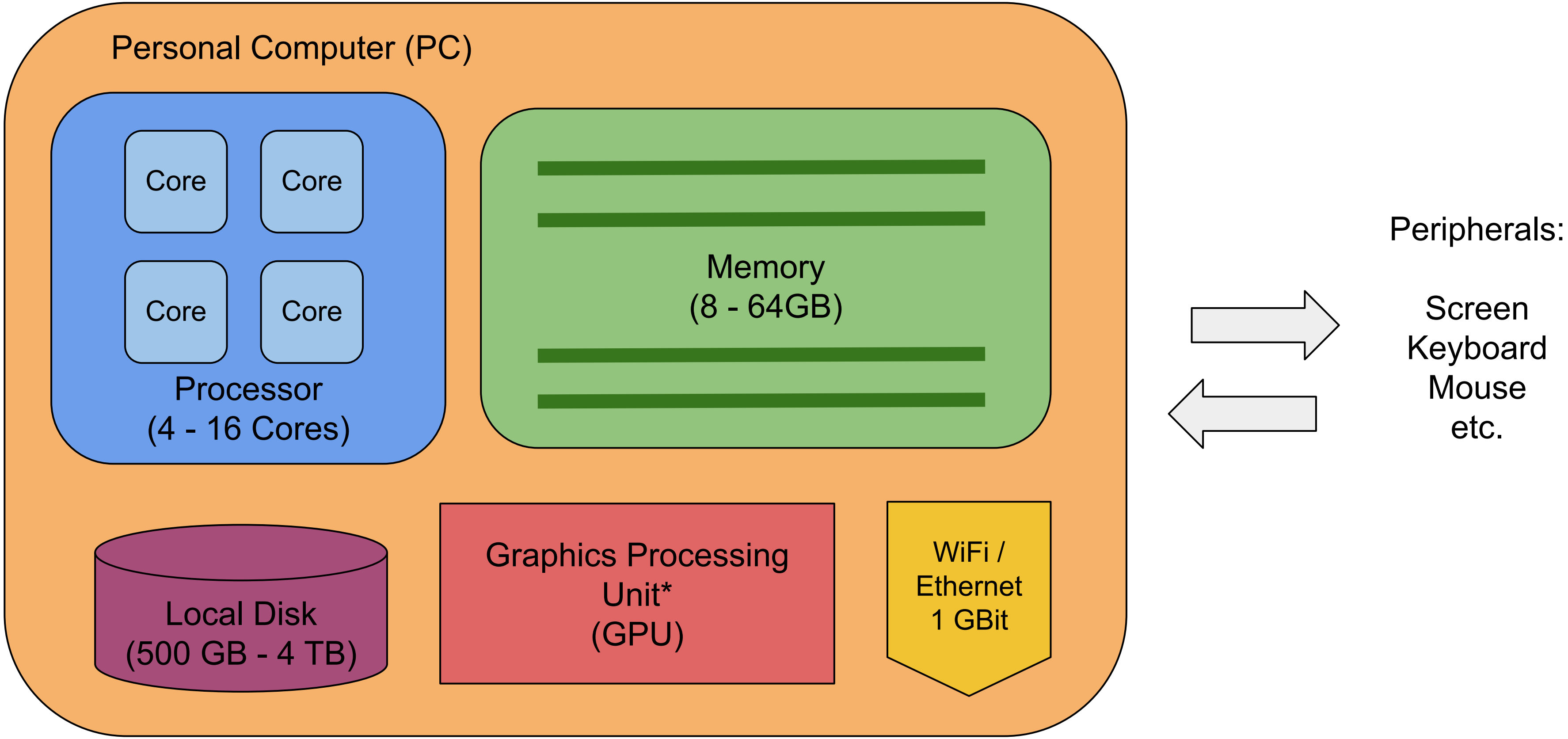
# Topics

- A Review of Computers
  - Basic Definitions
- What is the shell?
  - Basic Navigation
- Interacting with Files
  - Basic File Handling
- Finding Files
  - Getting What You Need
- Scripting
  - Getting the Same Results
- High Performance Computing (HPCs)
  - A Brief Introduction

# A Review of Computers

# Working on a personal computer (PC)

- Everyone is familiar with this.
    - You’re using one now.
- These machines are designed to facilitate the most common needs of the most people.
- Understanding how these machines work will make it easier to use modern data science tools.

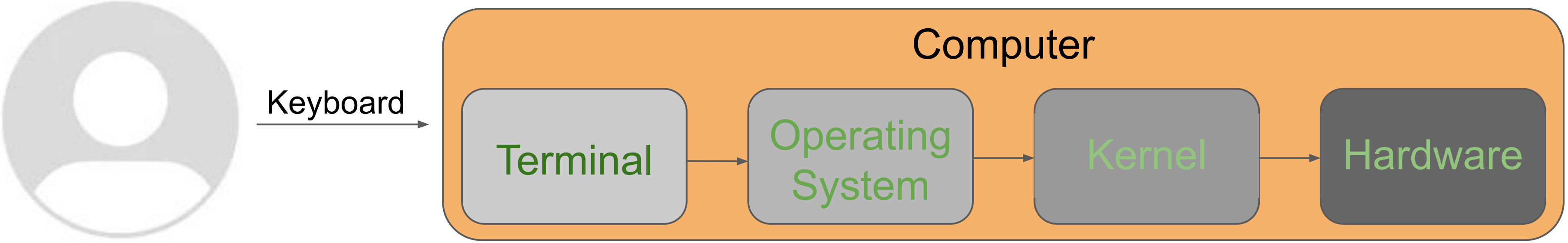



* not necessarily present in all systems (laptops)

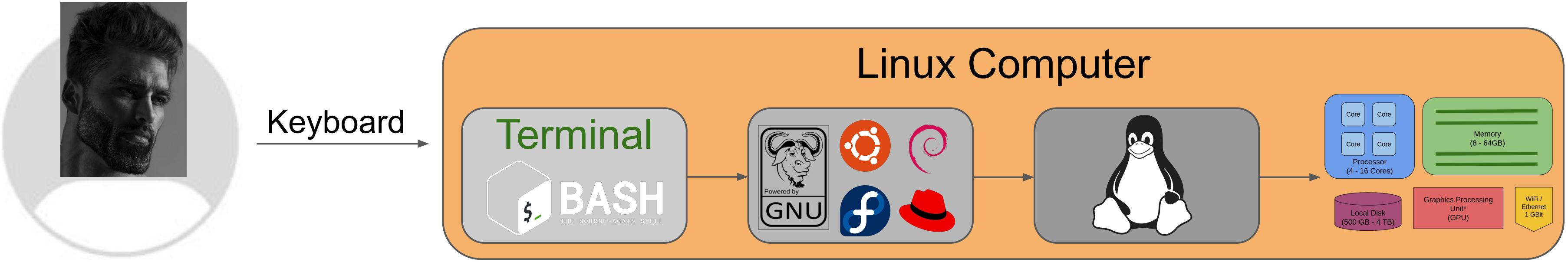
# Some Definitions

- Terminal
    - A program that will give you access to an interactive shell environment.
- Shell
    - The interactive program that allows you to run other programs or interact with files.
    - `Bash`, `Zsh`, `PowerShell`, etc.
- Operating System
    - The underlying program (*supervisor*) that coordinates the execution of programs.
    - Ubuntu, OSX, Windows
- Kernel
    - The set of function calls that translate an operating systems commands to hardware operations.
    - Linux, BSD, Windows

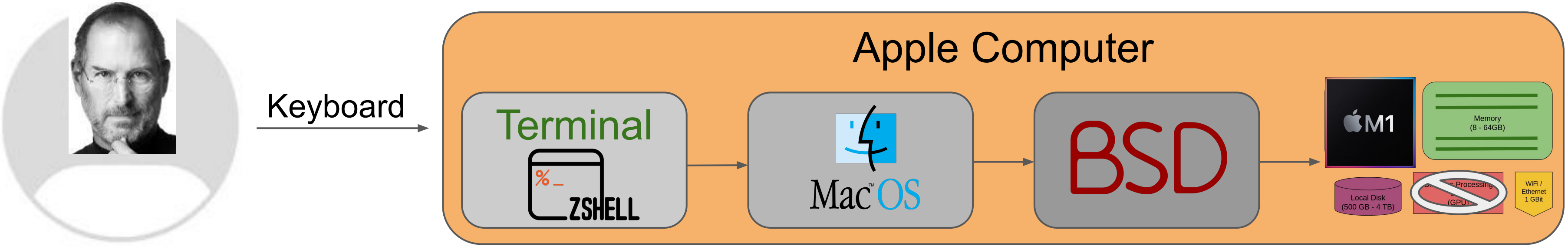
# Using a Computer



# Using a Linux Computer

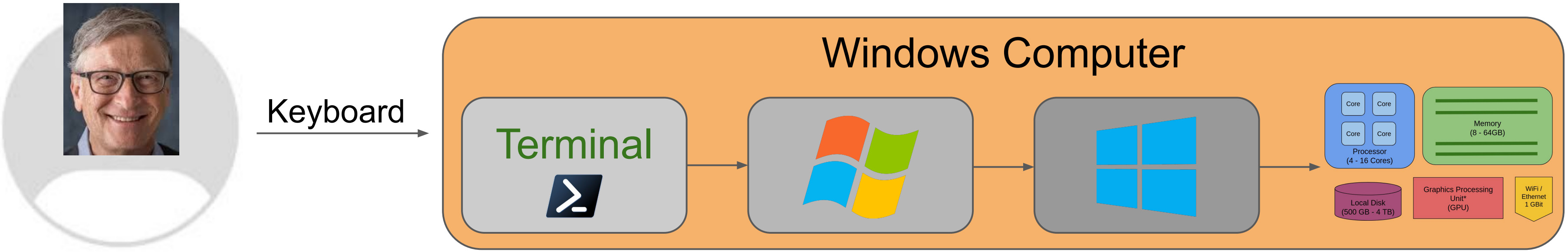


# Using a Mac

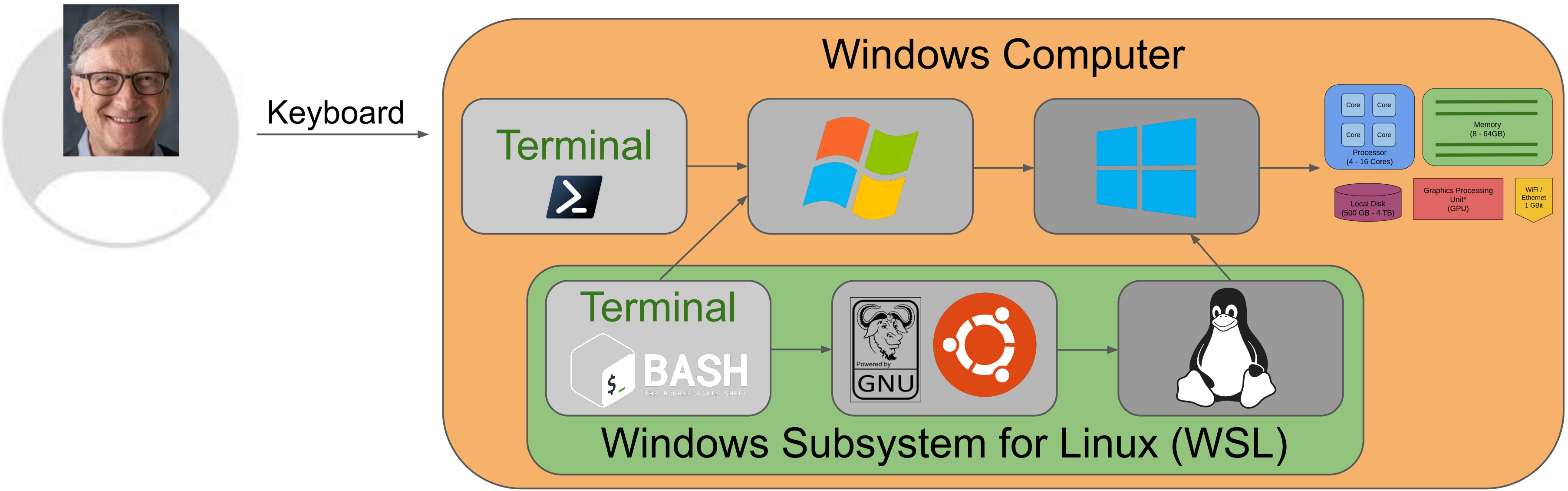


Keyboard
Apple Computer
Terminal
%_
ZSHELL
Mac OS
BSD
M1
Memory
(8 - 64GB)
Local Disk
(500 GB - 4 TB)
(GPU)
WiFi /
Ethernet
1 GBit

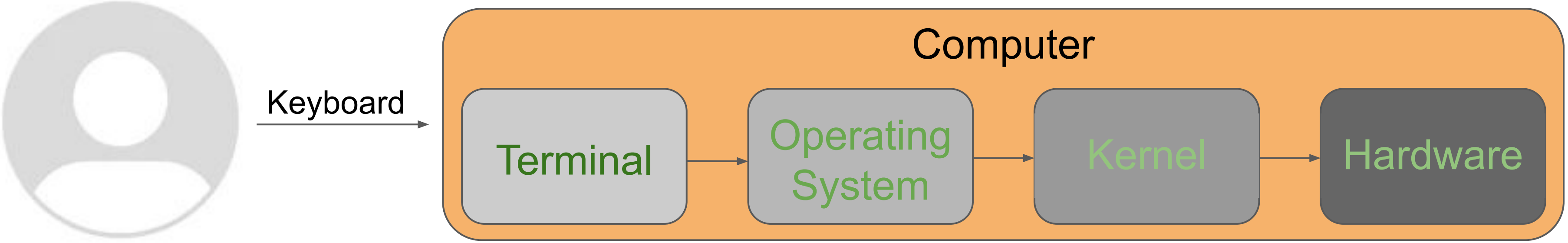
# Using a Windows Computer



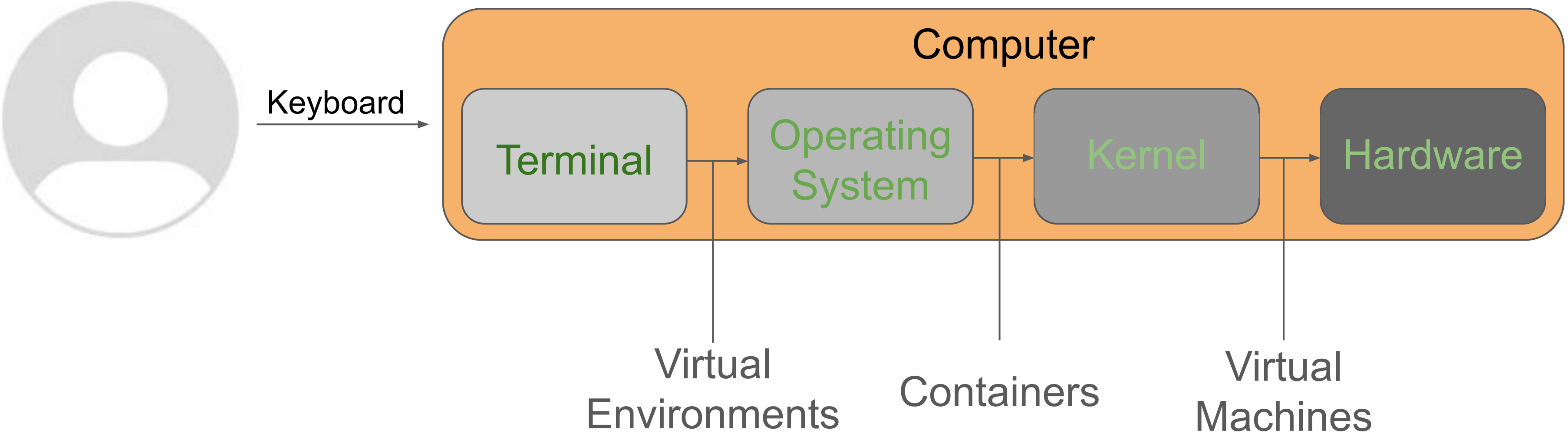
# Using Linux on Windows (?)



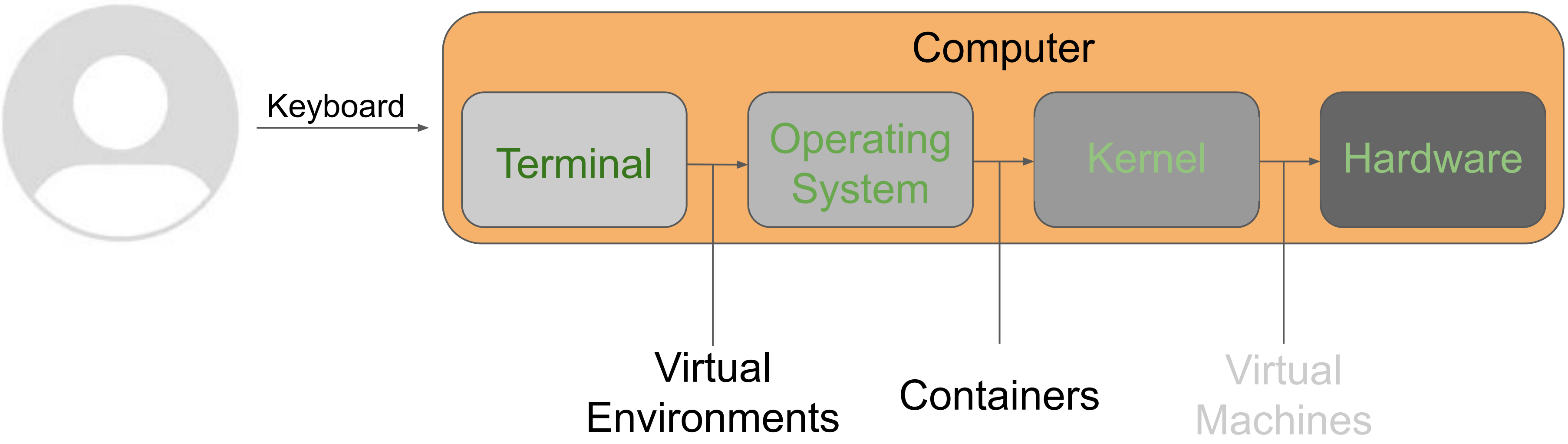
# Using a Computer



# Using a Computer for Reproducible Results



# Using a Computer for Reproducible Results



# Questions?

# What is the Shell Doing?

- A shell is a program that interprets user input into something a computer can understand.
  - Text-based interaction
- The command-line shell runs inside a terminal that lets you type commands.
  - A *command line interface* (**CLI**)
  - This is in contrast to a *graphical user interface* (**GUI**)
- By working in a CLI, you can develop a sequence of commands with a scripting language.

# There are many kinds of shells

- The most common shell is the Bourne Again Shell (Bash)
  - This is an updated version of the Bourne Shell (`sh`)
- There are shells with more human readable syntax.
  - C-shell (`csh`) / C-shell+ (`tcsh`)
- There are shells that are optimized for interactivity.
  - Z-shell (`zsh`) / friendly, interactive shell (`fish`)

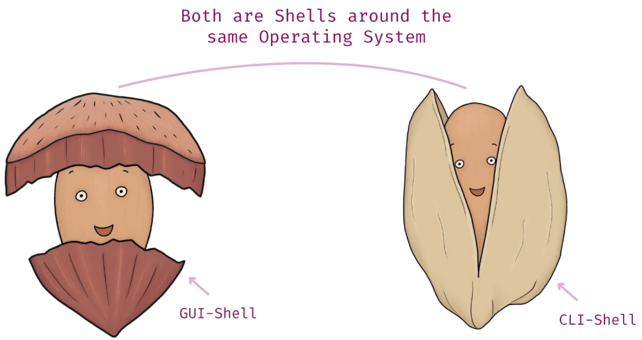
# What is the point of all the shells?

- They all have different strengths for different purposes.
  - Some have more interactive support.
  - Others have more complete scripting features.
- You will see different shell preferred throughout different software, too!
  - FSL is written in bash
  - AFNI recommends tcsh

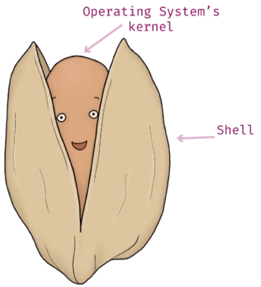
# Today’s Focus: Bash

- The most ubiquitous shell environment.
- It’s the default shell in most Linux systems.
  - Compute Canada / most HPCs.
  - This includes WSL.
  - OSX uses `zsh`, but bash is still available.
- The syntax for Bash is preserved across most common shells.

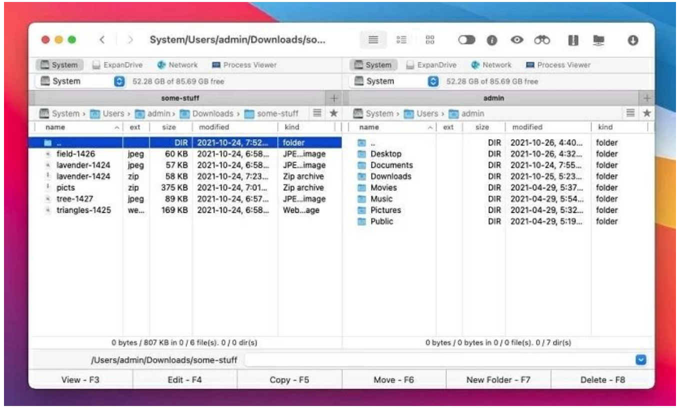
# But why use the shell at all?

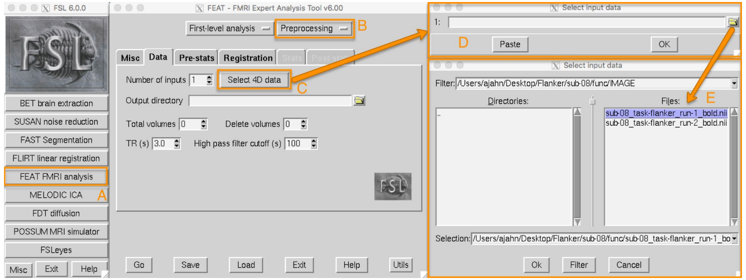


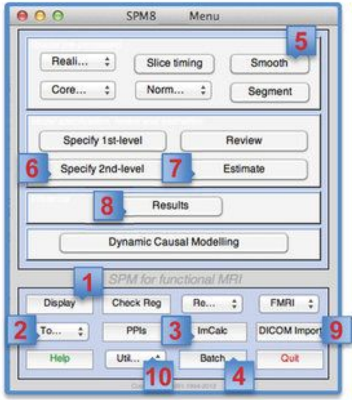

- A GUI is fine, but the shell is very powerful.
- Some tasks take many “clicks” in a GUI.
    - A shell script can automate this.
    - This can greatly ease reproducibility.
- You can sequence together the output of one program into another.
    - Perform different task automation or pipelining.
    - Most flexible way to combine different tools together.
- The script that was written can be (re-)executed easily.
    - Self-documenting
    - Easily generalizable
- On HPC systems, you’ll be interacting with the shell anyway.

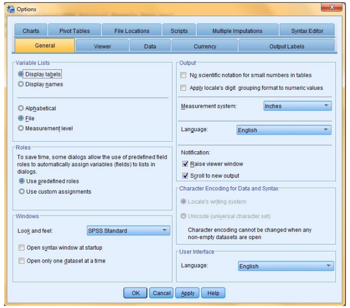


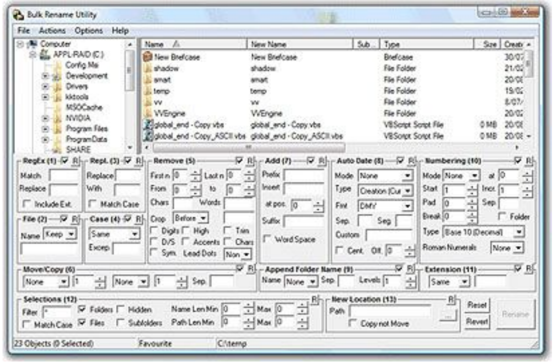
# Using a GUI is “easy”...




… but reproducible science is hard.

```
#!/bin/sh

# General FSL anatomical processing pipeline
#
#   Mark Jenkinson
#   FMRIB Image Analysis Group
#
#   Copyright (C) 2012 University of Oxford
#
#   Part of FSL - FMRIB's Software Library
#   http://www.fmrib.ox.ac.uk/fsl
#   fsl@fmrib.ox.ac.uk
#
#   Developed at FMRIB (Oxford Centre for Functional Magnetic Resonance
#   Imaging of the Brain), Department of Clinical Neurology, Oxford
#   University, Oxford, UK
#
```

```
Usage() {
    echo "Usage: `basename $0` [options] -i <structural image>"
    echo "       `basename $0` [options] -d <existing anat directory>"
    #echo "       `basename $0` [options] --list=<list of image names OR a text file>"
    echo " "
    echo "Arguments (You may specify one or more of):"
    echo "  -i <strucural image>         filename of input image (for one image only)"
    echo "  -d <anat dir>                directory name for existing .anat directory where this script will be run in place"
    echo "  -o <output directory>        basename of directory for output (default is input image basename followed by .anat)"
    #echo "  --list=<image list>          specifies a list of images to be averaged (either a comma separated list of image names with no
containing the individual image filenames)"
    echo "  --clobber                    if .anat directory exist (as specified by -o or default from -i) then delete it and make a new or
    echo "  --strongbias                 used for images with very strong bias fields"
    echo "  --weakbias                   used for images with smoother, more typical, bias fields (default setting)"
    echo "  --noreorient                 turn off step that does reorientation 2 standard (fslreorient2std)"
    echo "  --nocrop                     turn off step that does automated cropping (robustfov)"
    #echo "  --nobet                      turn off step that does brain extraction (BET or registration) - to use this the input image mus
```

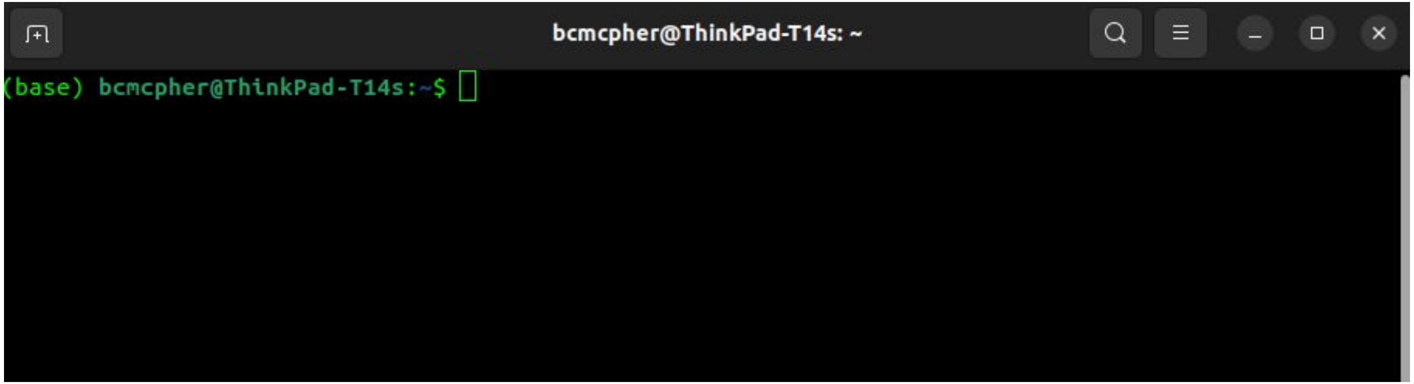
```
#### REORIENTATION 2 STANDARD
# required input: ${T1}
# output: ${T1} (modified) [ and ${T1}_orig and .mat ]
if [ $do_reorient = yes ] ; then
    date; echo "Reorienting to standard orientation"
    run $FSLDIR/bin/fslmaths ${T1} ${T1}_orig
    run $FSLDIR/bin/fslreorient2std ${T1} > ${T1}_orig2std.mat
    run $FSLDIR/bin/convert_xfm -omat ${T1}_std2orig.mat -inverse ${T1}_orig2std.mat
    run $FSLDIR/bin/fslreorient2std ${T1} ${T1}
fi


#### AUTOMATIC CROPPING
# required input: ${T1}
# output: ${T1} (modified) [ and ${T1}_fullfov plus various .mats ]
if [ $do_crop = yes ] ; then
    date; echo "Automatically cropping the image"
```

```
#### SKULL-CONSTRAINED BRAIN VOLUME ESTIMATION (only done if registration turned on, and segmentation done, and it is a T1 image)
# required inputs: ${T1}_biascorr
# output: ${T1}_vols.txt
if [ $do_reg = yes ] && [ $do_seg = yes ] && [ $T1 = T1 ] ; then
    echo "Skull-constrained registration (linear)"
    run ${FSLDIR}/bin/bet ${T1}_biascorr ${T1}_biascorr_bet -s -m $betopts
    run ${FSLDIR}/bin/pairreg ${FSLDIR}/data/standard/MNI152_T1_2mm_brain ${T1}_biascorr_bet ${FSLDIR}/data/standard/MNI152_T1_2mm_skull $
${T1}2std_skullcon.mat
    if [ $use_lesionmask = yes ] ; then
        run ${FSLDIR}/bin/fslmaths lesionmask -max ${T1}_fast_pve_2 ${T1}_fast_pve_2_plusmask -odt float
        # ${FSLDIR}/bin/fslmaths lesionmask -bin -mul 3 -max ${T1}_fast_seg ${T1}_fast_seg_plusmask -odt int
    fi
    vscale=`${FSLDIR}/bin/avscale ${T1}2std_skullcon.mat | grep Determinant | awk '{ print $3 }'`;
    ugrey=`$FSLDIR/bin/fslstats ${T1}_fast_pve_1 -m -v | awk '{ print $1 * $3 }'`;
    ngrey=`echo "$ugrey * $vscale" | bc -l`;
    uwhite=`$FSLDIR/bin/fslstats ${T1}_fast_pve_2 -m -v | awk '{ print $1 * $3 }'`;
    nwhite=`echo "$uwhite * $vscale" | bc -l`;
```

# Let’s Get Started

# Getting Started - The Prompt

- When the shell is first opened, you’ll be presented with a prompt.
  - This indicates the shell is waiting for an input.
- This can be customized by the user to provide more detail in a session.
  - We will not cover that today, but there are a lot of resources available online.



# What shell am I using?

- This is easy to check. Type:
  - `echo $0`
- It should display: `bash`
- To explicitly enter a Bash session, type
  - `bash`
  - `echo $0`
- This will put you in a Bash session.

# Bash Session

- Bash is a program that is running and interactive in the terminal.
- From a Bash session, you can launch a different shell environment.
    - Type `tcsh` to start a `tcsh` session, etc.
- You can also do this with Bash.
    - Your Bash session can run another interactive Bash session.
    - This is a *subprocess*.

# What is with the `$`?

- Things prefixed with `$` in bash are variables.
    - All programming languages have variables.
- When we want to reference a variable we need a `$` prefix.
- By default, your system / shell will have some variables already available. These are called **environment variables**.
    - `$SHELL` - your default shell - an environment variable
    - `$0` - the name of the running program, i.e. your active shell
    - `printenv` - print all the variables currently in your environment.

# Working within the Shell

- There is a lot of typing that happens when you’re working within a Terminal.
- The 2 most important tips are:
  - `Tab` Completion - Press the <Tab> to fill in available completions for your current prompt.
  - Command History - Press <Up> to cycle through the previously entered commands.
- There are also a bunch of keyboard shortcuts with `CTRL`
  - `CTRL + C` - end the current process (program) - but it won’t end a shell session
  - `CTRL + A / E` - go to the beginning / end of the current line
  - `CTRL + L` - clear the terminal
  - `CTRL + K` - delete to the end of the line
  - `CTRL + R` - search the command history
  - `CTRL + D` - exit the session (interactive process)
- If you find yourself repeating a process, check if there’s a shortcut to help!

# So where are we right now?

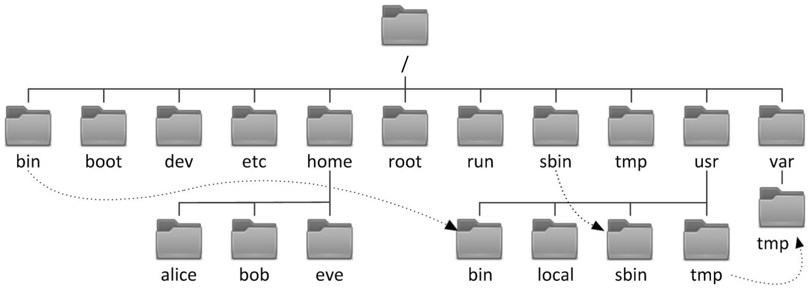
- When we open our terminal we are placed somewhere in the file system.
    - At any time while using a shell we are in exactly one place.
- Commands can read / write / operate on files wherever we are, so it’s important to know where we are!
- We can find our current working directory with the following command:
    - `pwd`
- Many bash commands are acronyms or abbreviations to try and help you remember them.
    - The above command, `pwd`, is an acronym for “*print working directory*”.

# Navigating Files and Directories

- The **filesystem** is how the operating system manages files and directories.
  - A **file** is a single document.
  - A **directory** is a folder that holds many documents.
- Generally, a new terminal will open with you in your home directory.
  - `~/` or `/home/$USER` or `/Users/$USER` (OSX)
- There are a lot of shell commands to create, inspect, rename, move, or delete files and directories.
  - File handling operations are maybe the most common reason to work in a shell!

# A quick tour of the Filesystem

- The top level directory is `/` (root)
  - Inside a path `/` is used as a separator between directories.
- Inside are several other directories:
  - `/bin` contains built-in (*binary*) programs
  - `/usr` is where miscellaneous program files are stored
  - `/home` (`~/`) is where personal user directories are kept
  - `/tmp` is for temporary files



# Your `/home` (`~/`) directory

- This is where many of your personal files are stored.
- There are many common directories here.
  - Documents
  - Downloads
  - Templates
  - etc.
- This is also where many of your configuration files are stored.
- On a personal PC, your work may be here as well.
- On a remote PC, you will probably have a project drive separate from your home directory for your work.

# What is in a directory?

- Now we can determine where we are (and know what that means).
  - But what is currently there?
- In order to view the contents of our current working directory, we need to *list* the contents *to* the *screen*.
  - `ls` - *list to screen*
- If we want to augment how `ls` displays this result, we can add **options** or **flags**. This will change the behavior of the program.
  - `ls -F` - *list to screen showing the type of files found.*
  - `ls -a` - *list to screen showing hidden files (most configuration files).*

# General Syntax of a Shell Command

- Options (arguments, switches, flags) change the behavior of a command.
- They generally start with a `-` or `--`
  - `-h` or `--help` are the most likely ways to get a quick breakdown of what a user can do.
  - `man ls` will open the manual entry for the command (not all programs have an entry).
  - `whatis` is a useful helper that might be easier to get started with.
- Flags are case sensitive
  - `ls -r` and `ls -R` do different things.
- For example, to list to screen the contents of our current working directory as a list that is sorted in order based on the size of the files:
  - `ls -lS`

# Moving Around

- At this point we can:
  - See where we are: `pwd`
  - See what files we have: `ls`
  - Understand the basic parts of a shell command.
  - Get help from the command line.
- Let’s *change directories* to a different location to do some work.
  - `cd ~/shell-course`

# Changing Directories

- By default, an interactive session starts in your home directory.
    - `~/`
- Typing `cd` without any arguments will take you back to your home directory.
    - The `~` is a shortcut for `$HOME` (`/home/$USER`)
- `cd -` will take you back to the last directory your were working in.

# Changing Directories

- Changing directories can be done with either **absolute** or **relative paths.**
- Absolute paths: `/home/bcmcpher/Documents/Project/learn-bash/demo.sh`
  - Are defined in reference to the root directory.
  - This is very precise, but also very verbose to type.
- Relative paths: `../learn-bash/demo.sh`
  - Are defined in reference to the current working directory: `./`
  - They do not begin with `/`
  - `../` refers to the directory above. This can be repeated (`../../../`)
    - `ls -a`

# Hidden Files

- Having discovered `..` and `.` in the current working directory, you may see other files.
- These hidden files are prefixed with `.` and commonly called dotfiles.
- They can be either **files** or **directories**.
- By default they are not displayed.
    - This is just a convention for convenience.
- Hidden files often contain configuration settings or version control records.

# Summary - Basic Navigation

- `echo` a statement or variable to the terminal as text.
    - `echo “Statement and Variable $VAR to print.”`
- `pwd` - “print working directory”
    - `pwd -P`
- `ls` - “list to screen”
    - `ls <PATH>` , default `.`
- `cd` - “change directory”
    - `cd <PATH>`, default `~/`

# Interacting with Files

# Creating Directories

- In order to start a new analysis, let's make a directory to store it.
  - `mkdir analysis`
- This will add a folder called `analysis` to your current working directory.
- Please follow good naming conventions in your files and folders:
  - DO NOT use < **s p a c e s** > of any kind.
  - Stick with letters, numbers, `-`, and `_`
  - DO NOT use special characters: `~!@#$%^&*(),`

# Creating Files

- In a folder where you want to create a new file, you can
    - Open a new file in VS Code / Emacs / Vim / etc.
    - Use a CLI tool (like `nano`) to create an new file.
    - `cat` (*con**cat**enate*) the output of files together. If called on a single file, it prints it.
    - `touch` a new file into existence.
- Generally, the shell works well with *plaintext* files.
    - *Binary* files will need a function to provide a text summary.

# nano

- `nano` is a useful CLI text editor.
- It will open within the terminal.
    - This is useful when working on remote systems that cannot open a new window.
- You don’t necessarily need to use this, but it’s helpful to remember it exists.
- The commands are always at the bottom of the screen.
    - `CTRL + KEY` _ is displayed as `^KEY`.

# Creating Files - Redirect

- Programs often print text to the terminal that we want to store.
- We can capture text being printed to the terminal by *redirecting* it.
    - `>` redirects output text to a file. It will overwrite (replace) whatever is in the file.
    - `>>` appends output text to a file. It will add to whatever currently exists in the text file.
- There are many advanced ways to redirect text - these are just the basics.
    - This is most common for creating logs.

# Moving Files and Directories

- Maintaining a deliberate and organized file structure sometime means we need to *move* files or even whole directories.
    - `mv <original> <new>`
- Importantly, renaming a file or folder is more accurately *moving* it to a new name in its current location.
- You can move multiple things at once to the last argument.
    - `mv file1.txt file2.txt new_folder`
- You can **glob** (`*`) or **wildcard** (`?`) individual characters in a filename for pattern matching.
    - `mv file* new_folder`
    - `mv file?.txt new_folder`

# Be Careful!

- `mv` is quite dangerous.
    - **Aliasing** `mv` to `mv -i` to add the *interactive* flag is a decent idea.
    - This will prompt you for a **yes / no** (y / n) anytime a file would be overwritten.
- Otherwise, `mv` will silently (and instantly) overwrite or delete files.
- **This can not be undone**. There is no Trash / Recycle bin in the terminal.
    - You can accidentally move all your subject results to the same file name and loose all the data, etc.

# Copying Files and Directories

- *Copy* (`cp`) is a lot like move, except the original and new files both exist.
- All the same glob / wildcard rules apply to `cp`.
  - Directories will need to be copied with `-r` (*recursive*).
- Importantly, `cp` does not care if a file already exists - it will move it again.
  - `rsync` will check and only move what it needs to - even between systems.
  - `rsync -av <source/> <destination>`

# `rsync` Examples

- Too much or not enough?

# Linking Files and Directories

- *Linking* (`ln`) is a lot like copy, except the file isn't copied.
    - This creates a *symbolic link* from the created *source* to the *target* file.
- A *link*, or shortcut, is created in the current directory to the target file.
- Opening the shortcut opens the linked file - there is no duplicate!
    - Changes in one "file" will be reflected in the "other".
- `ln -s <shortcut> <target-file-to-link>`

# Removing Files and Directories

- Eventually, you will just need to *remove* a file.
    - `rm <file>`
- Like *moving* (`mv`), there is no undo option when you `rm` a file.
    - THE SHELL DOES NOT HAVE A TRASH / RECYCLE BIN.
    - Consider *aliasing* `rm -i`
- Removing a directory needs the *recursive* flag (`rm -r`) in order to complete.
- There is also `rmdir`
    - This does not work if a folder is not empty - a safe way to try removing a directory if you're cleaning up a failed run.

# Summary - Basic File Handling

- `mkdir` - “make a directory”
  - `mkdir <new-directory>`
- `touch` - poke a file or make a new, empty one.
  - `touch <new-file>`
- `mv` - “move”
  - `mv <original-place-name> <new-place-name>`
- `cp` - “copy”
  - `cp <original> <new-copy>`
- `>`, `>>` - redirecting output
- `rsync` - “remote sync” the files in <source> to <destination>
  - `rsync -av <source> <destination>`
- `ln` - symbolic “link”
  - `ln -s <shortcut-linked-file> <the-original-file>`
- `rm` - “remove”
  - `rm <file>`
- `rmdir` - “remove directory”
  - `rmdir <empty-directory>`
- `nano`

# Finding Files

# Finding Files

- The filesystem can become quite complex, with many similarly named files and nested subdirectories.
- To display all the files in a directory, you can use:
  - `ls -R` (note the capital `R`!)
  - `tree` *(may not be installed)*
- To search and return file names with a specific pattern, you can use `find`
  - `find . -name “*.sh”`

# Finding Things Inside of Files

- Often enough, it is not the names of the file, but the contents of it that you want to search across.
- For this, we want to ***g***lobally search for a ***r***egular ***e***xpression and ***p***rint the matching lines.
  - `grep “find this pattern” text-file-001.txt`
- This will match patterns within the matching text files.
  - `grep “find this pattern” text-file-*.txt`

# Finding Things Inside of Files

- There are many basic descriptions of text files you cat get from the CLI
  - `head` - print the first `-n` lines of a file.
  - `tail` - print the last `-n` lines of a file.
  - `wc` - print the *word count* of the file; `-l` counts the number of lines.
  - `cat` - print the whole file.

# Passing Information - Pipes `|`

- One of the strengths of the shell is the ability to connect the output of one command into the next.
- This is done with the *pipe* character: `|`
- Connecting commands with a `|` doesn’t store the intermediate step beyond passing it as the input to the next command.
    - `command1 -flags arguments | command2 -flags arguments <implicit-output-of-command1>`
    - `ls -lF | head`
    - `cat text-file-001.txt | wc -l`
    - `find . -type f -name “subj*_output.txt” | sort | tail -n 15`

# Passing Information - Pipes `|` - Advanced

- You can pass information through multiple pipes.
- You can either work on the entire output (previous examples) or you can work on each output individually with `xargs`.
- `find . -type f -name “subj*_output.txt” | sort | ` **`xargs -n 1 -I {} tail -n 15 {}`**

# Summary - Finding Files

- `tree` will show a structured directory *tree* of all nested files and directories.
  - `tree <directory-to-show>`
- `find` will find all files in a location that match the pattern in the arguments.
  - `find <in-directory> -type f -name “*out.txt”`
- `grep` will look for a pattern within a file(s).
  - `grep “pattern” file-to-search`
- `head` will show the top lines of a file.
- `tail` will show the bottom lines of a file.
- `wc` will count the words (or lines) of a file.
- `sort` will sort the order of outputs.
- `xargs` will apply a command to each item in a list.

# Scripting

# Scripts and Variables

- The principal power of the shell is the ability to write reusable scripts.
- This is helpful if you want to:
  - Run the exact same commands or process again.
  - Keep a detailed record of the commands and parameters that produce an output.
  - Generalize a working analysis to new inputs and parameters.
  - Share or distribute a set of commands to other users.

# When do you write a Shell Scripts vs. a Python Script?

- 

# Creating a Shell Script

- A shell script is a *plaintext* file.
    - It can have any kind of extension (or none at all).
    - Conventionally you would use `.sh`
- At the top of the file is the *shebang* line
    - `#!/bin/bash`
    - This tells the system what kind of shell environment to run these commands.
        - `#!/bin/tcsh`
        - `#!/usr/bin/python3`

# Shell Variables - `$`

- Shell variables are set with `=`
  - There is no space around the equals - *it will not work with spaces*
    - `X=10`
    - `my_var=5`
- Variables are referenced with `$`
  - `echo $X`
  - `echo $my_var`
- The “strict” invocation of a variable uses `{ }`
  - `echo ${X}`
  - This is necessary if `_` is in the variables name.
- By convention, shell variables are:
  - `ALL_CAPS` if they do not change once they are set.
  - `lower_case` if they change or are reassigned (like within a loop).

# Shell Variables and Environment Variables

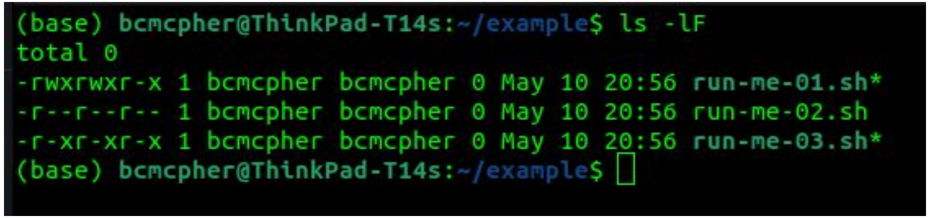
- Environment Variables are available system-wide.
  - Regular variables are only available within their specific session / script (their *scope*).
- You `export` variables to find them in your environment.
  - `export MY_VAR=5` ; `echo $MY_VAR`
  - Scripts find environment variables within their scope.
- To remove an environment variable, you `unset` it.
  - `unset $MY_VAR`

# Running a shell script

- Run a shell script by invoking it.
  - `./run-me.sh`
  - `bash run-me.sh`
- However, this is unlikely to work on a new script
  - You need to set the *file permissions*.

# File Permissions

- Files are accessible at different levels of functionality to different users.
    - You can give different users (or groups of users) access to your files.
- The kinds of actions that can be taken on a file depend on their permissions.
    - **Read** (r) - view contents
    - **Write** (w) - change contents
    - **Execute** (x) - run the program




- Permissions are applied separately to the ***user***, ***group***, and ***others***.
- In order to run a new shell script, the file that contains the script must be marked (flagged) as executable.
    - `chmod +x run-me.sh`

# How does the shell know where to find commands?

- The system looks for executables along a default **Path**.
  - This is an environment variable defined by the system.
- By adding a folder with executables to the path those programs become available to the users terminal sessions.
- Otherwise, a relative or absolute path to the script must be invoked.
- `echo $PATH`

# How does the shell know where to find commands?

- The order of folders on the paths is the priority commands will be run.
  - The first program found is the first program run.
  - Virtual environments *prepend* dependencies to your path so they are found first and used.
- In setting up different tools, your path may do unexpected things.
- Use `which` to find the path of a program you are calling.
  - This can help you identify the specific tool your using and identify if your `$PATH` has become corrupt.

# Modifying the Path

- As part of installing software, you may need to manually add it to your `$PATH`.
- To do this for every Bash session moving forward, modify your `~/.bashrc`
  - When making changes, it may be safer to manually set them up until you are sure they work.
- Your `~/.bashrc` (run commands) runs every bash session you start.
  - It defines some variables by default.
- You can append / prepend new tools to your path or add variables to initialize in your default environment.
  - `export PATH=/opt/new/tool:$PATH`
  - `export SUBJECTS_DIR=/home/$USER/freesurfer/v6.0.1`

# Modifications to Commands

- In addition to modifying the path, the `~/.bashrc` is useful for setting up defaults tags for commands, or ***aliases***.
  - WARNING - you reap what you tweak. Make small, incremental changes and check them.
- An **alias** lets you define specific flags or entirely new commands.
  - `` alias la=`ls -a` ``
  - `` alias mv=`mv -i` ``
  - `` alias beluga=`ssh -X bcmcpher@beluga.computecanada.ca ` ``
- You can save a lot of time by customizing your environment to your specific needs.

# Summary - Finding Files

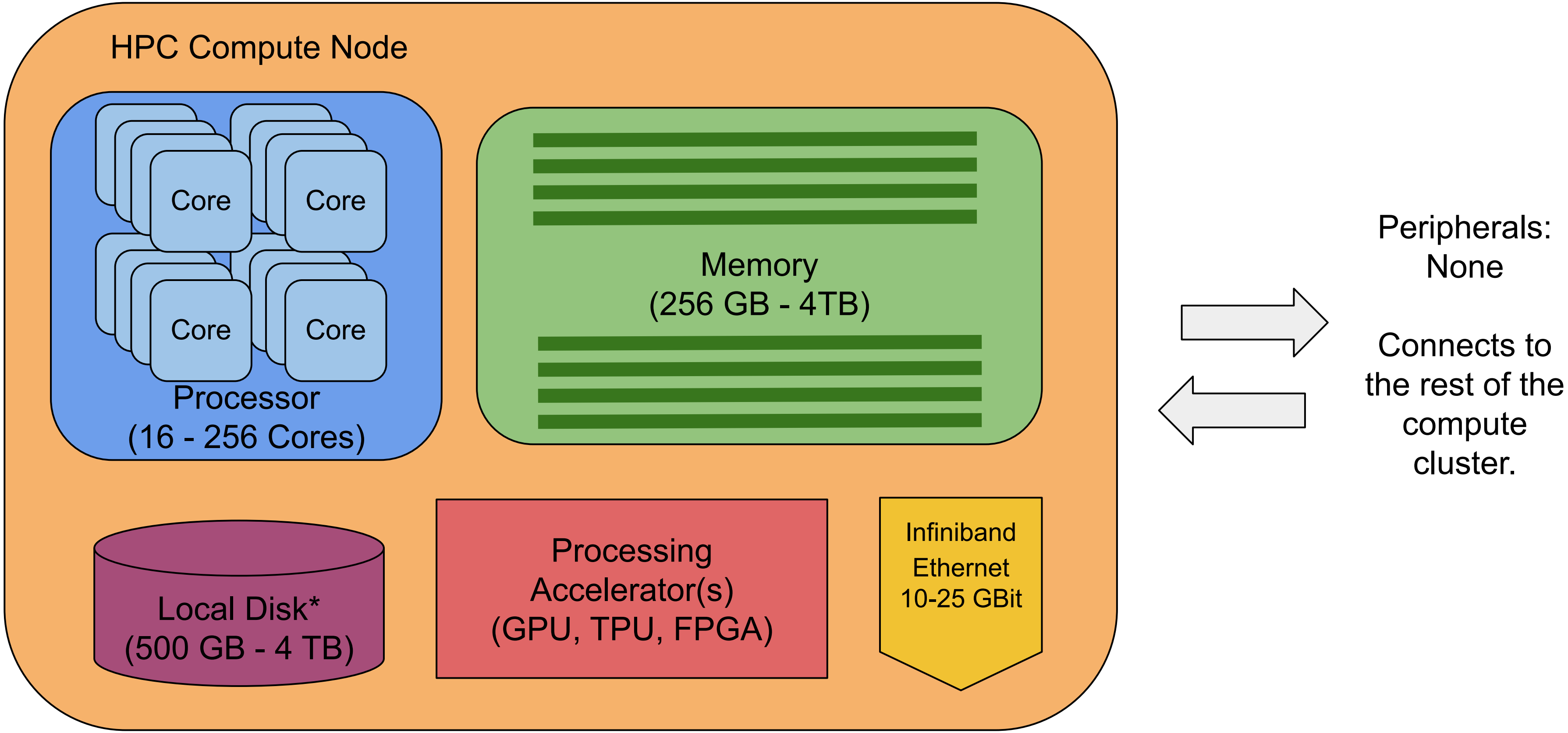
- Shebang line - `#!/bin/bash`
- Variables - `$`
    - Isolated to the current session.
- Environment Variables - `export MY_VAR` ; `unset MY_VAR`
    - Can be found system-wide.
- `chmod`
    - File Permissions - `rwx`
- The `$PATH`
- The `~/.bashrc`
- `alias`

# Questions?

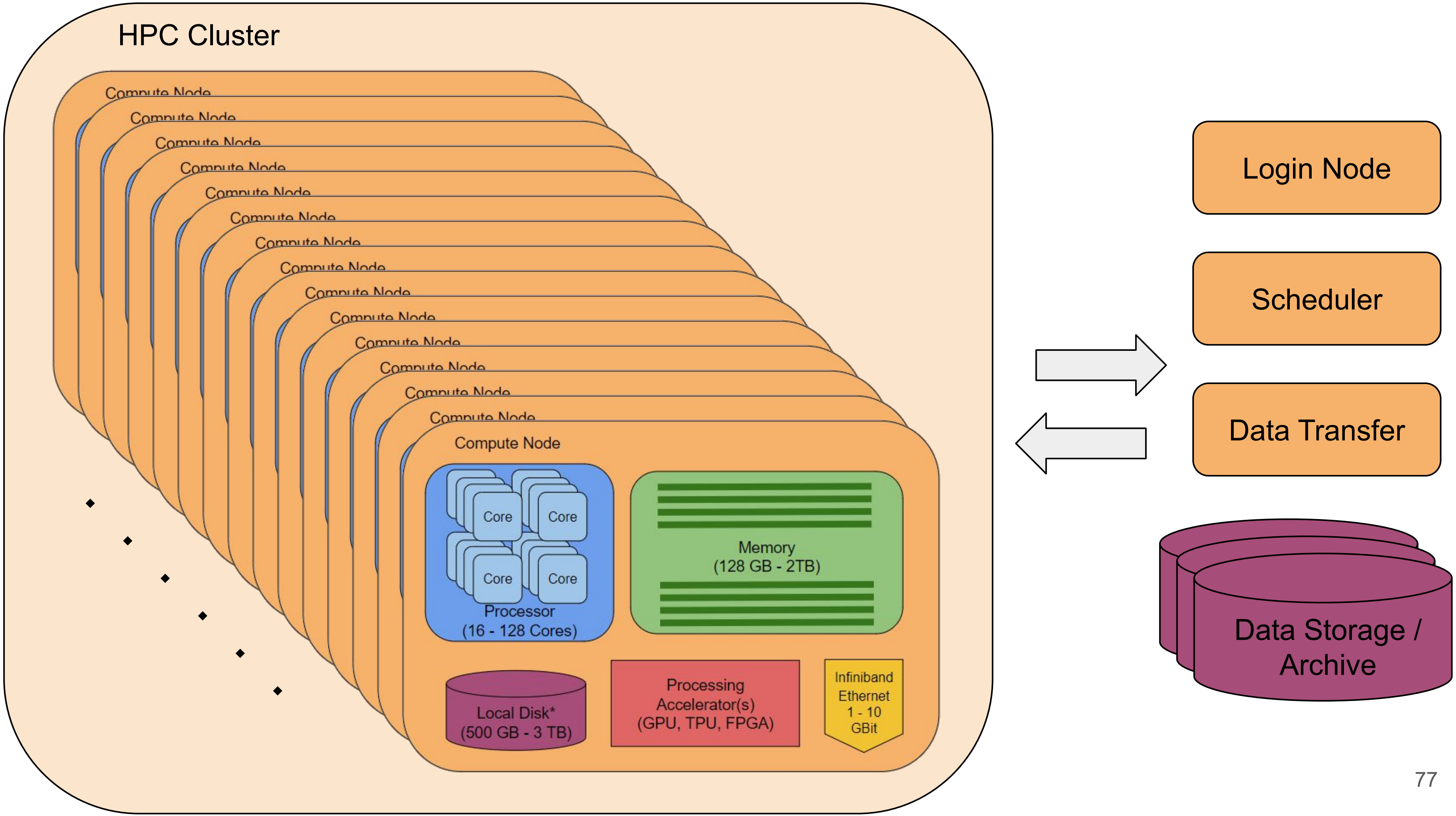
# Brief Introduction to HPC

# Working on an HPC

- These are not “personal” systems, they are multi-user.
- You do not interact with them directly, but over a network connection in a shell.
- The interaction with your analysis entails:
    - Transferring data / results.
    - Managing your analysis “jobs”.
- There is minimal “interactive” computing performed on a HPC.
    - Typically little to no visualization is performed on HPC systems.
- They almost exclusively run Linux.




* it may be less and / or have higher access speed


HPC Cluster
Compute Node
Core
Core
Core
Core
Processor
(16 - 128 Cores)
Memory
(128 GB - 2TB)
Local Disk*
(500 GB - 3 TB)
Processing
Accelerator(s)
(GPU, TPU, FPGA)
Infiniband
Ethernet
1 - 10
GBit
Login Node
Scheduler
Data Transfer
Data Storage /
Archive

# The Anatomy of a Computer

## PCs

- 1 CPU
  - 4 - 16 cores
  - 3.8 - 5.0 GHz
- 8 - 64 GB RAM
- 500 GB - 4 TB Storage
- 1 GPU
  - This may be built into the CPU
- WiFi / 1 GBit Ethernet
- x86 (maybe ARM) architecture
- Single-User System
- Good for most general usage
  - This is likely where you will develop your analysis.

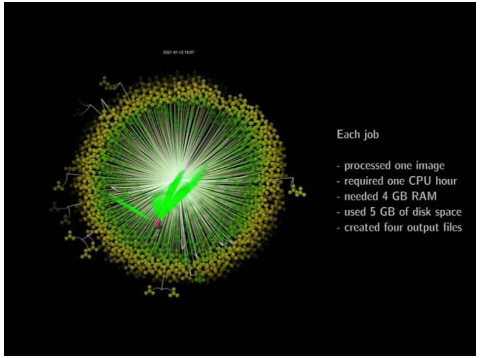
## HPCs

- Multiple CPUs
  - 16 - 256 cores per CPU
  - 2.4 - 3.5 GHz
- 256 GB - 4 TB RAM
- 500 GB - 4 TB Storage
  - Not for long term storage
- 0 - 4 Processor Accelerators
  - GPUs w/ double precision CUDA
  - TPUs, FPGAs, Coprocessors, etc.
- 10 - 25 GBit Ethernet, Infiniband
- Different architectures (x86, PPC, Cray)
- Multi-User System
- Good for high throughput / large models.

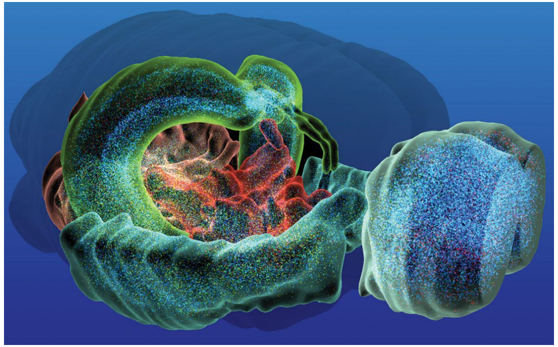
# The advantages of a HPC over a PC

- The ability to handle large datasets in a more time efficient manner.
- The option to “**Scale Out**” or “**Scale Up**” an analysis.
    - **Scale Out**: the ability to analyze independent parts of a dataset simultaneously
        - Independent permutations / cross-validations / simulations.
        - Different subjects through the same preprocessing preparations.
            - “Embarrassingly Parallel”
            - High Throughput Computing
    - **Scale Up**: create a larger single instance of computing resources to run a larger model, estimation, or analysis.

# Scaling Out




# Scaling Up



[UKBB high-throughput replication of over 40K subjects]

# Job Scripts

demo_subj01.slurm

```
#!/bin/bash

#SBATCH --job-name=demo_subj01                        # job name
#SBATCH --nodes=1                                     # run on a single node
#SBATCH --ntasks=1                                    # run on a single CPU
#SBATCH --cpus-per-task=1                             # run on a single core
#SBATCH --mem=1gb                                     # job memory request
#SBATCH --time=00:05:00                               # time limit hrs:min:sec
#SBATCH --error=./jobs/logs/demo_subj01_%j.err        # standard error from job
#SBATCH --output=./jobs/logs/demo_subj01_%j.out       # standard output from job
#SBATCH --account=def-jbpoline                        # define your affiliation

## ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
## MODIFY THE RESOURCE REQUSTS ABOVE FOR YOUR JOB

## MODIFY THE CODE BELOW TO CALL YOUR ANALYSIS
## vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv

## your job script generalized to a subject ID
bash ./analysis.sh subj01
```

# Commands for working with Jobs

- sbatch
    - Submit a job to the scheduler.
    - sbatch job.slurm
- squeue
    - Monitor the status of jobs on the cluster.
    - squeue -u <USERNAME>
- scancel
    - Cancels a job that hasn’t completed.
    - scanel <JOBID>
- sacct
    - Provides a summary of the jobs your have recently submitted.
    - sacct
- srun
    - Start an interactive job to test that your job will work correctly.
    - srun --nodes=1 --ntasks-per-node=1 --time=01:00:00 --pty bash -i

There are a lot of resources available for learning about HPCs online and in your lab. Find a process that works for you and ASK how others have solved similar problems!

Thanks!