# Week 4

DATS 6450 – FOUNDATIONS OF COMPUTER SCIENCE

# Class Overview

- Quiz 1
- Review and answer questions on Week 3 lecture
- Linux Commands continued
  - Finding out more about files
  - Types of Linux Commands
  - Aliasing
  - Permissions
  - I/O & Redirection
  - Piping & Filter Commands

# Quiz 1

- 30 minutes
- No Laptops/Phones
  - You should only have the (printed) quiz and a pen/pencil on your desk

# More info on files

- You can find out what type of data a file contains with the **file** command:
  - Syntax: **$ file <file_name>**
  - Example: **$ file my-textfile.txt**

| File Type | Description | Viewable as text? |
|---|---|---|
| ASCII text | The name says it all | yes |
| Bourne-Again shell script text | A **bash** script | yes |
| ELF 32-bit LSB core file | A core dump file (a program will create this when it crashes) | no |
| ELF 32-bit LSB executable | An executable binary program | no |
| ELF 32-bit LSB shared object | A shared library | no |
| GNU tar archive | A tape archive file. A common way of storing groups of files. | no, use **tar tvf** to view listing. |
| gzip compressed data | An archive compressed with **gzip** | no |
| HTML document text | A web page | yes |
| JPEG image data | A compressed JPEG image | no |
| PostScript document text | A PostScript file | yes |
| RPM | A Red Hat Package Manager archive | no, use **rpm -q** to examine contents. |
| Zip archive data | An archive compressed with **zip** | no |

# Viewing the contents of text files

- The **cat** command allows users to display the contents of a text file to the console:
  - Syntax: **$ cat <file_name>**
  - Example: **$ cat file1.txt**
- Sometimes the file is too large for the text to fit on the screen and it will be difficult to read. This problem can be solved by using the **less** command which displays the text file one page at a time:
  - Syntax: **$ less <file_name>**
  - Example: **$ less file1.txt**
    - To *scroll forward*, use the **space bar**
    - To *scroll backward*, use the letter **b**
    - To *exit*, press **q**

# Viewing the contents of text files

- Often the first thing we do when we get a new dataset is look at the first few lines of the file to see the column headers and the last few lines for some more example observations.
- We can do this on the command line the same way we do this in R and Python: the **head** and **tail** commands.
- Head and tail show the first/last 10 lines of a text file
  - Example: **$ head iris.csv**
- You can change the number of lines to show with the **–n** flag
  - Example: **$ head –n 5 iris.csv**

# Viewing the contents of text files

- The next thing we may want to know is how many observations we have (rows/lines) or the total number of words and characters.
- We can find out the *word count* with **wc**:
  - Syntax: **$ wc <file_name>**
  - Example: **$ wc file1.txt**
- By default, it gives us the number of characters, words, and lines in a file
- You can make it print only one of these using `-c`, `-w`, or `-l` respectively.

# Types of Commands

- We've used a number of commands so far, but what are commands?
- There are 4 main types of commands:
  1. **Executable program**
     - Most of the files in */usr/bin* are compiled binaries written in languages like C, and C++ or other programs written in scripting languages like Perl, Python, etc.
  2. **Shell Built-ins**
     - Commands that are internally built into the shell. For example, *cd* is a shell built-in.
  3. **Shell function**
     - Shell scripts incorporated into the *environment*. We will cover scripting and environment management next class
  4. **Alias**
     - Commands that you can define yourselves. They are built from other commands and data scientists often make their own shortcuts with aliases.

# Types of Commands

- So how do we figure out the type of command we are using?
  - We can do this with the **type** command:
    - Example: **$ type cd**

- To locate commands on our computer, we use the **which** command:
  - Example: **$ which ls**

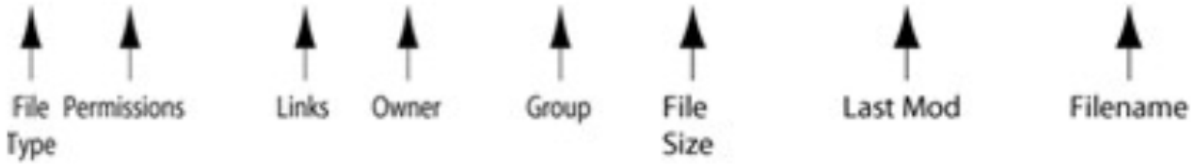
# Demo

# Alias

- An alias is nothing more than a keyboard shortcut that we can make to save time working at the command line
- We saw before in the demo that **type ll** displayed:
  - ll is aliased to `ls -al'
  - This is because we created an alias to perform **ls –al** every time we type **ll**
  - We can create the following alias with
    - **$ alias ll="ls -al"**
    - Note we don't use spaces before or after the equal sign
  - You can get rid of the alias with **unalias**
    - **$ unalias ll**

# Permissions

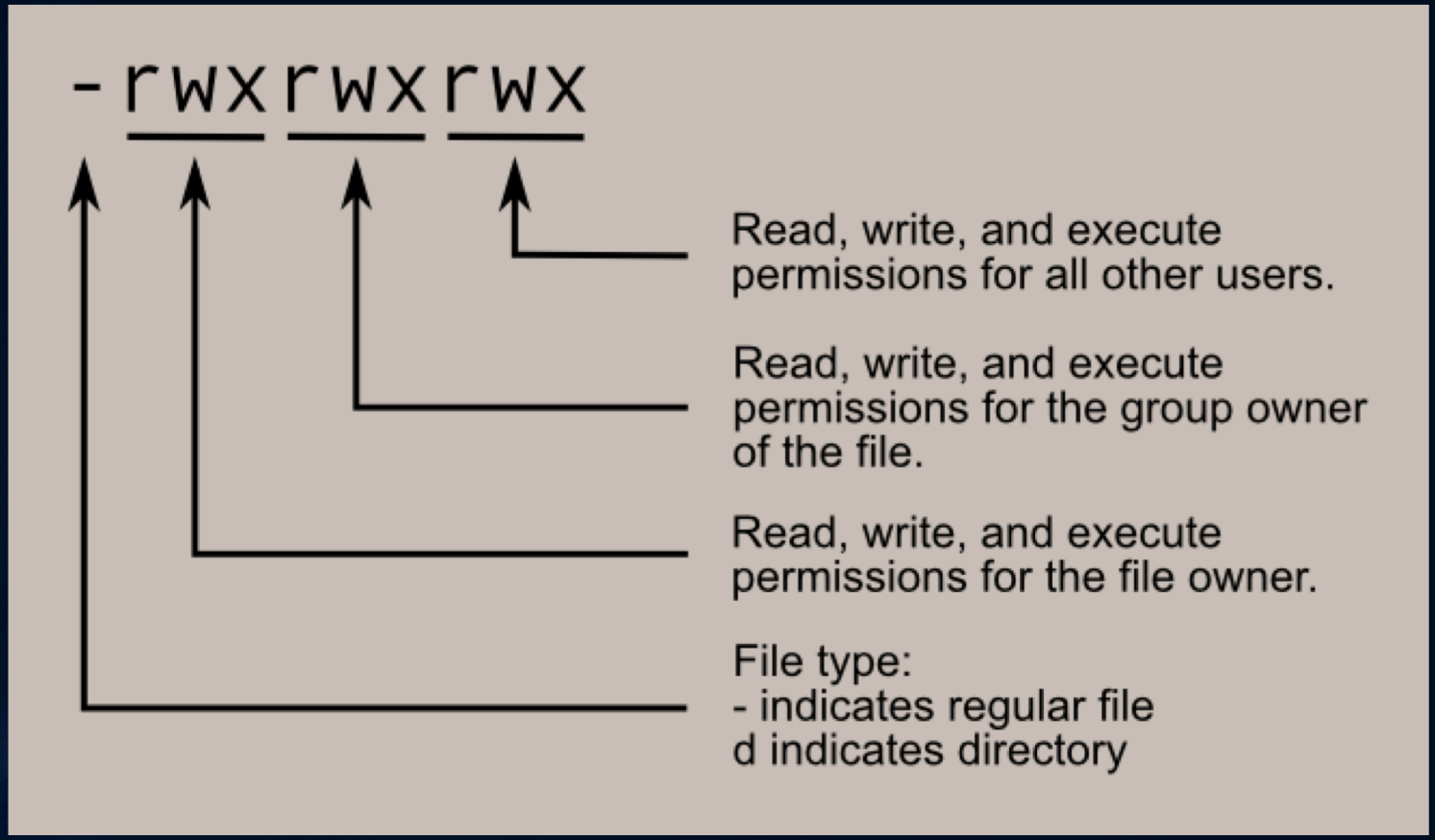
```
% ls -al
total 126
drwxr-xr-x 13 ege  csci 1024 Apr 26 15:49 .
drwxr-xr-x 15 root root  512 Apr 24 15:18 ..
-rwxr--r--  1 ege  csci  885 Dec  2 13:07 .login
-rwx------  1 ege  csci  436 Apr 12 11:59 .profile
drwx------  7 ege  csci  512 May 17 14:11 330
drwx------  3 ege  csci  512 Mar 19 13:31 467
drwx------  2 ege  csci  512 Mar 31 10:16 Data
-rw-r--r--  1 ege  csci   80 Feb 27 12:23 quiz.txt
```



# Permissions

- Unix-like operating systems differ from other computing systems in that they are not only *multitasking* but also *multi-user*.
- This means that more than one user can be operating the computer at the same time.
  - For example, if your computer is attached to a network, or the Internet, remote users can log in via ssh (secure shell) and operate the computer.
  - In order to make this practical, a method had to be devised to protect the users from each other.
    - For example, you don't want users to have access to other users' files

# Permissions



# Permissions

- You should think of the permission settings as a series of bits.
  - This is called the *octal* method.

  - **rwx rwx rwx = 111 111 111**
  - **rw- rw- rw- = 110 110 110**
  - **rwx --- --- = 111 000 000**

  - **rwx = 111 in binary = 7**
  - **rw- = 110 in binary = 6**
  - **r-x = 101 in binary = 5**
  - **r-- = 100 in binary = 4**

# Permissions

- To *change* the permissions of a file of a file/directory you wish to *modify*, you can use the **chmod** command
  - Syntax: **$ chmod <xxx> <file>**
  - Example: **$ chmod 777 file.txt**
    - Makes file.txt readable, writable, and executable to everyone (fully open)
  - Example: **$ chmod 400 my_awskey.pem**
    - Makes my_awskey.pem only readable by the owner (who created the file)
    - This protects you from someone stealing your key and accessing your aws instance.

# Permissions

- You may need to become the **superuser**, a special user account used for system administration, to perform a given operation.
- The **superuser** is the user who has all rights or permissions (to all files and programs) in all modes (single- or multi-user)
- To execute a command as the superuser, you need to prepend **sudo** to the desired command.
  - Example: **$ sudo some_command**
  - You will be prompted for your user password after entering the command

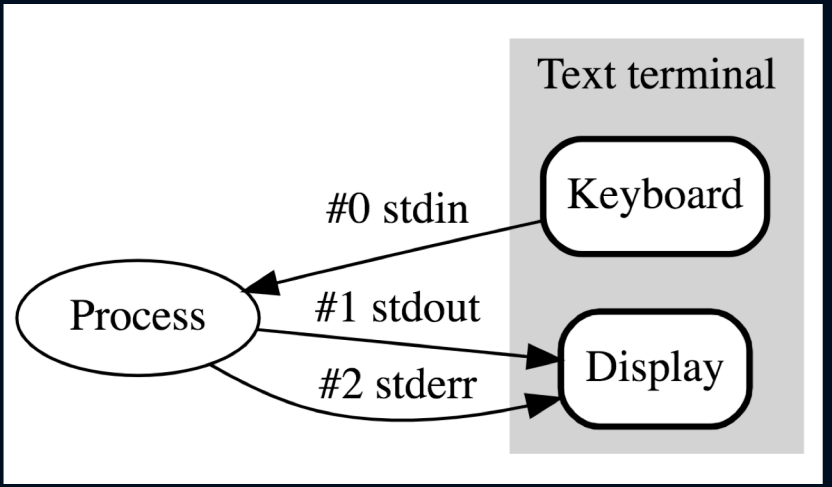
# Permissions

- To *change* the *owner* of a file, you can use the **chown** command, but you will most likely have to use **sudo**:
  - Syntax: **$ sudo chown <user> <file>**
  - Example: **$ sudo chown Guest file.txt**
- To *change* the *group ownership* of a file, you can use the **chgrp** command, but you will have to be the owner of the file to change the group ownership:
  - Syntax: **$ sudo chgrp <group> <file>**
  - Example: **$ sudo chgrp  file.txt**

# Demo

# Standard Input, Output, Error

- In computer programming, **standard streams** are preconnected input and output communication channels between a computer program and its environment when it begins execution.
- The *three* input/output (I/O) connections are
  1. standard input (stdin)
  2. standard output (stdout)
  3. standard error (stderr)
- When a command is executed via an interactive shell, the streams are typically connected to the text terminal on which the shell is running, but can be changed with redirection or a pipeline.



# I/O Redirection

- One of the coolest functions of the command line is I/O Redirection
  - We can *redirect* the *input* and *output* of commands to and from files, as well as chain multiple commands together into command *pipelines*
- Most of the commands we have run so far produce output of some kind
  - There are *two* types of output:
    1. Results - the data/output the program or command was designed to produce
    2. Status and error messages
  - By default, both standard output (*stdout*) and standard error (*stderr*) are linked to the screen (and not saved to disk)
  - Programs also take input from a stream called standard input (*stdin*) which is, by default, attached to the keyboard

# I/O Redirection

- I/O redirection allows us to change where output goes and where input comes from.
- As previously mentioned, output normally goes to the screen and input comes from the keyboard, but with I/O redirection, we can change that.
- What is the point of redirecting output? It's often useful to store the output of a command in a file.
- To redirect standard output to another file instead of the screen, we use the ">" redirection operator followed by the name of the file.
- Syntax: **$ command > outputfile.txt**
- Example: **$ ls -l /usr/bin > binary_list.txt**

# I/O Redirection

- A single “>” will always create a new file and re-write from the beginning
- If we want to append to an existing file, we use the “>>” redirection operator.
- Syntax: **$ command >> outputfile.txt**
- Example: **$ ls -l /sbin >> binary_list.txt**

# I/O Redirection

- What will happen if we try redirect a command that fails?
- Example: **$ ls -l /bin/usr > binary_list.txt**
- The error is still printed to the terminal, and if we examine the file it will be empty.
- But why?

# I/O Redirection

- As previously mentioned, both *stdout* and *stderr* default to stream to the terminal and since calling ***ls*** on a non-existing file sends *only stderr* **(**not *stdout)* it is still displayed!
- While we have referred to the first three file streams as *stdin*, *stdout* and *stderr*, the shell references them internally as file descriptors 0, 1 and 2, respectively.
- The shell provides a notation for redirecting files using the file descriptor number, and since ***stderr*** is file descriptor number 2, we can *redirect stderr* with the following notation:
  - Syntax: **$ command 2> outputfile.txt**
  - Example: **$ ls -l /bin/usr 2> binary_list.txt**
- This time the error is not displayed to the terminal, and if we open binary_list.txt we will see the error:
  - *ls: /bin/usr: No such file or directory*

# I/O Redirection

- What if we wish to capture *both stdin* and *stdout?*
- Recent versions of bash provide a streamlined method for *combined redirection:*
  - Syntax: **$ command &> outputfile.txt**
  - Example: **$ ls -l /bin/usr &> binary_list.txt**
- You can also *append both stdin and stderr* with the following:
  - Syntax: **$ command &>> outputfile.txt**
  - Example: **$ ls -l /bin/usr &>> binary_list.txt**

# Demo

# Pipelines

- A pipeline is a shell feature that allows *commands* to read data from *stdin* and send to *stdout*
- The *stdout* of one command can be *piped to* the *stdin* of another using the pipe operator "|"*:*
  - Syntax: **$ command1 | command2**
  - Example: **$ ls -l /usr/bin | less**
- But what is the difference between ">" and "|"?
  - Simply put, the *redirection operator* connects a command with a file while the *pipeline operator* connects the output of one command with the input of a second command

# Pipelines

- A lot of people mistakenly try to just redirect a command to another command (I get it)
- This can be pretty bad….

Here is an actual example submitted by a reader who was administering a Linux-based server appliance. As the superuser, he did this:

```
# cd /usr/bin
# ls > less
```

The first command put him in the directory where most programs are stored and the second command told the shell to overwrite the file `less` with the output of the `ls` command. Since the `/usr/bin` directory already contained a file named "less" (the `less` program), the second command overwrote the `less` program file with the text from `ls` thus destroying the `less` program on his system.

The lesson here is that the redirection operator silently creates or overwrites files, so you need to treat it with a lot of respect.

# Pipelines + Filters

- The real power of pipelines becomes apparent when users can perform very complex operations on data by simply stringing commands together.
- Commands that are used in this fashion are referred to as *filters.*
- Filters take input, modify it, and then output it.
  - Syntax: **$ command1 | command2 | command3**
  - Example: **$ ls /bin /usr/bin | sort | less**
- The first command specifies two lists (one for each directory) but because we piped it to *sort*, it returns a single, sorted list, and displays the results one page at a time since that list is piped to *less.*

# Common Filter Commands

| Program | What it does |
|---|---|
| **sort** | Sorts standard input then outputs the sorted result on standard output. |
| **uniq** | Given a sorted stream of data from standard input, it removes duplicate lines of data (i.e., it makes sure that every line is unique). |
| **grep** | Examines each line of data it receives from standard input and outputs every line that contains a specified pattern of characters. |
| **fmt** | Reads text from standard input, then outputs formatted text on standard output. |
| **pr** | Takes text input from standard input and splits the data into pages with page breaks, headers and footers in preparation for printing. |
| **head** | Outputs the first few lines of its input. Useful for getting the header of a file. |
| **tail** | Outputs the last few lines of its input. Useful for things like getting the most recent entries from a log file. |
| **tr** | Translates characters. Can be used to perform tasks such as upper/lowercase conversions or changing line termination characters from one type to another (for example, converting DOS text files into Unix style text files). |
| **sed** | Stream editor. Can perform more sophisticated text translations than **tr**. |
| **awk** | An entire programming language designed for constructing filters. Extremely powerful. |

# Wildcards

- Since the shell uses filenames so much, it provides special characters to help you rapidly specify groups of filenames.
- These special characters are called *wildcards*.
- Wildcards allow you to select filenames based on patterns of characters.

Summary of wildcards and their meanings

| Wildcard | Meaning |
|---|---|
| * | Matches any characters |
| ? | Matches any single character |
| [*characters*] | Matches any character that is a member of the set *characters*. The set of characters may also be expressed as a *POSIX character class* such as one of the following:<br>POSIX Character Classes<br>**[:alnum:]** Alphanumeric characters<br>**[:alpha:]** Alphabetic characters<br>**[:digit:]** Numerals<br>**[:upper:]** Uppercase alphabetic characters<br>**[:lower:]** Lowercase alphabetic characters |
| [!*characters*] | Matches any character that is not a member of the set *characters* |

# Wildcard Examples

Examples of wildcard matching

| Pattern | Matches |
|---|---|
| `*` | All filenames |
| `g*` | All filenames that begin with the character "g" |
| `b*.txt` | All filenames that begin with the character "b" and end with the characters ".txt" |
| `Data???` | Any filename that begins with the characters "Data" followed by exactly 3 more characters |
| `[abc]*` | Any filename that begins with "a" or "b" or "c" followed by any other characters |
| `[[:upper:]]*` | Any filename that begins with an uppercase letter. This is an example of a character class. |
| `BACKUP.[[:digit:]][[:digit:]]` | Another example of character classes. This pattern matches any filename that begins with the characters "BACKUP." followed by exactly two numerals. |
| `*[![:lower:]]` | Any filename that does not end with a lowercase letter. |

# grep

- The *grep* command is quite possibly the most common text processing command you will use.
- *grep* stands for *general regular expression parser.*
- It allows you to search files for characters that match a certain pattern.
- What if you wanted to know if a file existed in a certain directory or if you wanted to see if a string was found in a file?

# grep

- The *grep* command takes options, a pattern to match, and files to search in:
  - Syntax: **$ grep [options] PATTERN [FILES]**
  - Example: **$grep "Falls" county_fips.txt**

| Option | Meaning |
|---|---|
| `-c` | Rather than print matching lines, print a count of the number of lines that match. |
| `-E` | Turn on extended expressions. |
| `-h` | Suppress the normal prefixing of each output line with the name of the file it was found in. |
| `-i` | Ignore case. |
| `-l` | List the names of the files with matching lines; don't output the actual matched line. |
| `-v` | Invert the matching pattern to select nonmatching lines, rather than matching lines. |

# Demo

# Next Class

- Writing Shell Scripts
  - Text Editors
  - Variables
  - Control Flow
  - Shebang
  - Shell Functions
  - Making Executables
- Config Files
- Aliasing Continued