

Week 5

DATS 6450 – FOUNDATIONS OF COMPUTER SCIENCE

Class Overview

- Review and answer questions on Week 4 lecture
- Linux Commands continued
 - I/O & Redirection
 - Piping & Filter Commands
 - grep
 - cut

Recall from last time

- **cat** – (concatenate) reads files sequentially and writes their contents to standard output (the terminal – more on this shortly)
- **less** – display contents of a file to the terminal, allowing users to page through forward and backward with “space” and “b”, respectively
- **head** – display first –n lines to the terminal
- **tail** – display last –n lines to the terminal
- **wc** – count the number of words (-w) characters (-c) and lines (-l) in a file

Recall from last time

- **file** – tells you what type of data is in the file (text readable or not)
- **type** – tells you which type of command

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.

- **which** – prints the path/location of an executable that would have been executed if this argument had been entered into the shell

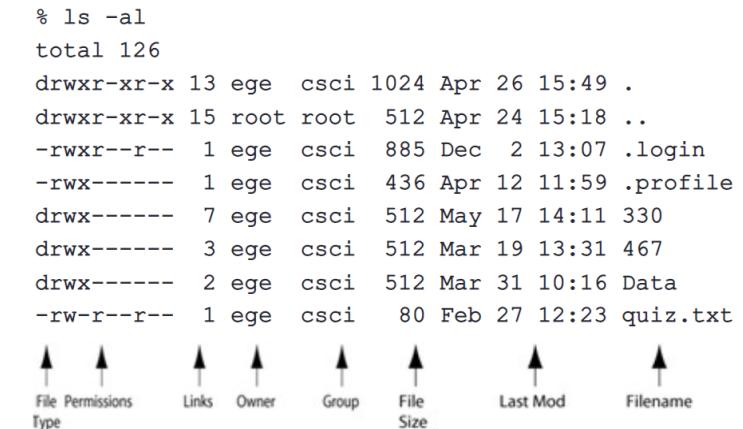
Recall from last time

- **alias** – a keyboard a keyboard shortcut that we can make to save time working at the command line
- To change the permissions of a file or 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.

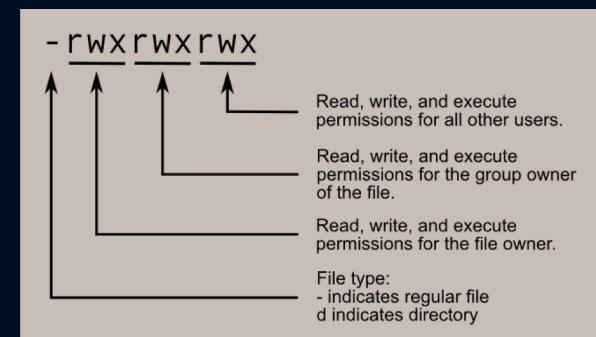
Recall from last time

- **chmod** – modify the permissions of a file or a file/directory
- **sudo** – (**superuser**) – the user who has all rights or permissions (to all files and programs) in all modes
- **chown** – change the owner of a file, (requires sudo)
- **chgrp** – change the group owner of a file

```
% 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
```



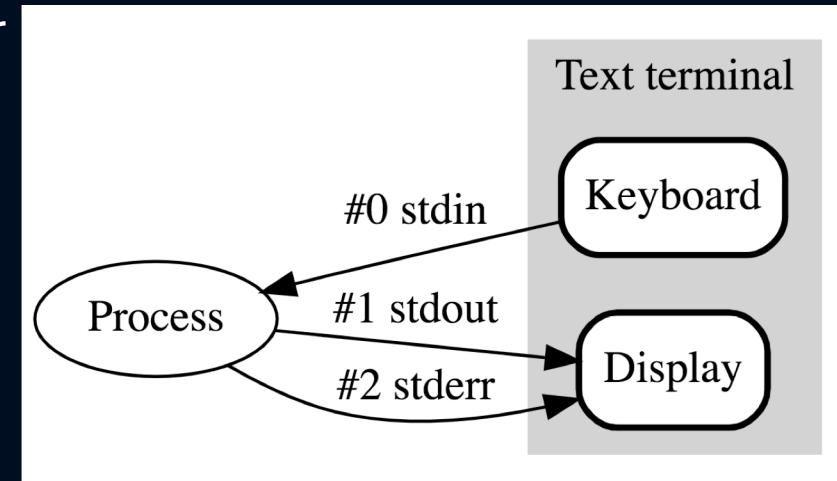
Binary	Decimal
rwx rwx rwx 111 111 111	777
rw- rw- rw- 110 110 110	666
rwx ---- 111 000 000	700
r--- --- 100 000 000	400



Week 5 Lecture

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 >> outfile.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

[hackerrank](#)

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
<u>sort</u>	Sorts standard input then outputs the sorted result on standard output.
<u>uniq</u>	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).
<u>grep</u>	Examines each line of data it receives from standard input and outputs every line that contains a specified pattern of characters.
<u>fmt</u>	Reads text from standard input, then outputs formatted text on standard output.
<u>pr</u>	Takes text input from standard input and splits the data into pages with page breaks, headers and footers in preparation for printing.
<u>head</u>	Outputs the first few lines of its input. Useful for getting the header of a file.
<u>tail</u>	Outputs the last few lines of its input. Useful for things like getting the most recent entries from a log file.
<u>tr</u>	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).
<u>sed</u>	Stream editor. Can perform more sophisticated text translations than <u>tr</u> .
<u>awk</u>	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
[<i>characters</i>]	Matches any character that is a member of the set <i>characters</i> . The set of characters may also be expressed as a <i>POSIX character class</i> such as one of the following: <small>POSIX Character Classes</small> [:alnum:] Alphanumeric characters [:alpha:] Alphabetic characters [:digit:] Numerals [:upper:] Uppercase alphabetic characters [:lower:] Lowercase alphabetic characters
[! <i>characters</i>]	Matches any character that is not a member of the set <i>characters</i>

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.

cut

- While **head** and **tail** allow you to select rows from a text file, you can use the **cut** command to select columns
- When you call the **cut** command, you need to specify the *fields* (columns) you want to extract and the *delimiter* that separates those columns
 - The **-f** flag is used to denote *fields*
 - The **-d** flag is used to denote *delimiters*
 - Example: **\$ cut -f 2 -d , data.csv**
 - This will take the second column of the dataset separated by a comma

SORT

- The sort program sorts the contents of standard input, or one or more files specified on the command line, and sends the results to standard output. (Typically used with pipes and usually before **uniq**)
- Prints the lines of its input or concatenation of all files listed in its argument list *in sorted order*.
- Sorting is performed on each line starting with the first character in the line.
- Blank space is the default field separator.
- A few options (there are many more):
 - -r : reverse the sort order
 - -n: sort according to numerical value
 - -f: ignore case (case insensitive)
 - -k: sort by key # (example later)
 - -b: ignore leading blanks - calculates sorting based on the first non-whitespace character on the line.

sort

- Example: sort the results of the du command to determine which packages that come with the anaconda distribution are the largest users of disk space. (Normally, the du command lists the results in pathname order).
- **\$ du -s /anaconda3/pkgs/* | sort -nr | head -n 10**
- By piping the output to sort, and using the -nr options, we produce a reverse numerical sort, with the largest values appearing first in the results. This sort works because the numerical values occur at the beginning of each line.
- But what if we want to sort a list based on some value found within the line?
 - For example, the results of an **ls -l**:

SORT

- I know that `ls` has a “-s” option for sort, but if we ignore that and look at the output of `ls -l`, we can pipe the output to sort to order our list by filesize as well

- `$ ls -l`

```
-rw-r--r-- 1 brentskoumal staff 3965921 Sep 24 21:47 adult.csv
-rw-r--r-- 1 brentskoumal staff 3974305 Sep 24 21:18 adult.data
-rw-r--r-- 1 brentskoumal staff 3965921 Sep 24 21:44 adult2.csv
-rw-r--r-- 1 brentskoumal staff          0 Sep 24 21:39 adult_sample.csv
-rw-r--r-- 1 brentskoumal staff        125 Sep 18 21:07 brents_text.txt
-rw-r--r--@ 1 brentskoumal staff 90975 Sep 19 17:36 county_fips.txt
-rw-r--r-- 1 brentskoumal staff        564 Sep 24 21:07 email_list.txt
-rw-r--r-- 1 brentskoumal staff         30 Sep 25 20:26 foo.txt
-rw-r--r--@ 1 brentskoumal staff 6246 Sep 24 21:55 group_export_7908.csv
-rw-r--r--@ 1 brentskoumal staff 24814 Sep 24 21:55 group_export_7908_by_course.csv
-rw-r--r-- 1 brentskoumal staff        140 Sep 24 21:27 header.csv
-rw-r--r-- 1 brentskoumal staff       193 Sep 24 20:21 redirected_output.txt
```

- `$ ls -l | sort -nr -k 5 | head`

- Many uses of sort involve the processing of tabular data, such as the results of the `ls` command above. If we apply database terminology to the table above, we would say that each row is a record and that each record consists of multiple fields, such as the file attributes, link count, filename, file size and so on. `sort` is able to process individual fields. In database terms, we are able to specify one or more key fields to use as sort keys. In the example above, we specify the `n` and `r` options to perform a reverse numerical sort and specify `-k 5` to make `sort` use the fifth field as the key for sorting.

uniq

- **Finding duplicates with uniq**
- With the uniq command you can find adjacent repeated lines in a file. uniq takes several flags, the more useful ones being:
 - **uniq -c**: which adds the repetition count to each line;
 - **uniq -d**: which only outputs duplicate lines; And
 - **uniq -u**: which only outputs unique lines.
- However, uniq is not a smart command. Repeated lines will not be detected if they are not adjacent. Which means that you first need to sort the file.

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.

curl

- curl is a tool to transfer data from or to a server, using one of the supported protocols (DICT, FILE, FTP, FTPS, GOPHER, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMTP, SMTPS, TELNET and TFTP). The command is designed to work without user interaction.
 - Example:
 - `$ curl https://repo.continuum.io/archive/Anaconda3-5.1.0-Linux-x86_64.sh > install.sh`
 - `$ bash install.sh`

sed

- The **sed** command is a unix utility that parses and transforms text
- At its core sed is a stream editor that operates on a line-by-line basis. It excels at substitutions, but can also be leveraged to do some very powerful stuff.
- The most basic sed command consists of `s/old/new/g`. This translates to search for old value, replace all occurrences in-line with new. Without the `/g`, our command would terminate after the first occurrence on the line.
 - it reads text, line by line, from an input stream or file, into an internal buffer called the *pattern space*. Each line read starts a *cycle*. To the pattern space, sed applies one or more operations which have been specified via a *sed script*.
 - The sed script can either be specified on the command line (`-e` option) or read from a separate file (`-f` option)
 - Syntax: **\$ sed "s/<string to replace>/<string to replace it with>/g" <source_file> > <target_file>**
 - Example: **\$ sed -i " 's/\([0-9]\),\([0-9]\)/\1\2/g' data.txt**

Demo

The Environment

- The shell maintains a body of information during our shell session called the *environment*.
- Data stored in the environment is used by programs to determine facts about the system's configuration.
- While most programs use *configuration* files to store program settings, some programs will also look for values stored in the environment to adjust their behavior.

What's Stored in the Environment?

- The shell stores *two* basic types of data in the environment:
 - *Environment Variables*
 - *Shell Variables*
 - Shell variables are bits of data placed there by bash and environment variables are everything else
- In addition to variables, the shell also stores *aliases* and *shell functions*.

Seeing What's in the Environment

- To see environment variables:
 - `$ printenv | less`
- To see environment and shell variables:
 - `$ set | less`
- To list the value of a specific variable:
 - `$ printenv USER`
- To see the contents of a variable:
 - `$ echo $HOME`
- To list all aliases:
 - `$ alias`

Expansion – How the Shell Sees Things

- Did you notice how we had to use **\$USER** to be able to print the value of the **USER** variable?
 - This is an example of *expansion*.
- Expansion is a process where we enter a command and it is actually *expanded* into something else before the shell acts on it
 - Example: `$ echo this is a test`
 - Example: `$ echo *`
 - Why didn't *echo* print “*”?
 - The simple answer is that the shell expands the “*” into something else (in this instance, the names of the files in the current working directory) before the *echo* command is executed

Expansion – How the Shell Sees Things

- Pathname Expansion
 - The mechanism by which *wildcards* work is called *pathname* expansion
 - This is what we saw on the previous slide with “*”
- Tilde Expansion
 - This is why when used at the beginning of a word, the tilde expands into the name of the home directory of the named user, or if no user is named, the home directory of the current user
- Brace Expansion
 - Allows us to create multiple text strings from a pattern containing braces
 - Example: `$ echo Number_{1..5}`
 - `Number_1 Number_2 Number_3 Number_4 Number_5`

Expansion – How the Shell Sees Things

- When might this be helpful?

```
[me@linuxbox ~]$ mkdir Photos
[me@linuxbox ~]$ cd Photos
[me@linuxbox Photos]$ mkdir {2007..2009}-{01..12}
[me@linuxbox Photos]$ ls
2007-01 2007-07 2008-01 2008-07 2009-01 2009-07
2007-02 2007-08 2008-02 2008-08 2009-02 2009-08
2007-03 2007-09 2008-03 2008-09 2009-03 2009-09
2007-04 2007-10 2008-04 2008-10 2009-04 2009-10
2007-05 2007-11 2008-05 2008-11 2009-05 2009-11
2007-06 2007-12 2008-06 2008-12 2009-06 2009-12
```

Expansion – How the Shell Sees Things

- Parameter Expansion
 - This is the type of expansion associated with having to put a **\$** in front of **USER** earlier to extract the value of the variable.
 - The shell uses parameter expansion so it knows whether you want a filename, value of a variable, etc.

Environment Variables

- Environment variables are usually stored as upper case
- A few of the most important ones to know are:

Variable	Contents
HOME	The path of the user's home directory
PWD	Present working directory; same value returned by pwd command
USER	Your username
OLDPWD	The previous working directory
SHELL	The name of the shell program being used
PATH	A colon-separated list of directories that are searched when you enter the name of a executable program.

- Try running: **\$ echo \$PATH**

How is the Environment Established?

- When we log on to the system, the *bash* program starts, and reads a series of configuration scripts called *startup files*, which define the default environment shared by all users.
- This is followed by more startup files in our home directory that define our personal environment.
- The exact sequence depends on the type of shell session being started. There are two kinds:
 - login shell session
 - One in which the user is prompted for username and password
 - non-login shell session
 - Typically occurs when we launch a terminal session in the GUI

Startup Files

- Login shells read:

File	Contents
/etc/profile	User's personal startup file. Extend or override global settings
~/.bash_profile	User's personal startup file. Extend or override global settings
~/.bash_login	If ~/.bash_profile isn't found, then bash tries to read this
~/.profile	If ~/.bash_profile and ~/.bash_login aren't found, then bash tries to read this

- Non-login shells read:

File	Contents
/etc/bash.bashrc	User's personal startup file. Extend or override global settings
~/.bashrc	User's personal startup file. Extend or override global settings

How does the shell find commands?

- Remember when we showed that the **ls** program was located at **/bin/ls** by running **which ls**?
 - So how does the computer know how to find the **ls** command when we don't supply the path?
 - The computer searches a list of directories that are contained in the **PATH** variable
- Did you get asked when downloading Anaconda if you wanted to add it to your **PATH** variable?
 - Saying "yes" adds the following to your **~/.bash_profile**:
 - **\$ export PATH="/Users/michaelarango/anaconda3/bin:\$PATH"**
 - Note: the *export* command tells the shell to make the contents of **PATH** available to child processes of this shell.

Shell Scripts

- A *shell script* is a file containing a series of commands (just like a Python script)
- There are three things we need to do to create and run shell scripts:
 1. Write the script
 - Shell scripts are text files, so you're going to want a good text editor.
 2. Make the script executable
 - You need to tell the computer that the text file is a program and change the permissions to allow execution
 3. Put the script somewhere the shell can find it
 - Add it to a directory in your PATH

Text Editors

- Several text editors exist

Name	Description	Interface
vi , vim	The granddaddy of Unix text editors, vi , is infamous for its difficult, non-intuitive command structure. On the bright side, vi is powerful, lightweight, and fast. Learning vi is a Unix rite of passage, since it is universally available on Unix-like systems. On most Linux distributions, an enhanced version of the traditional vi editor called vim is used.	command line
Emacs	The true giant in the world of text editors is Emacs by Richard Stallman . Emacs contains (or can be made to contain) every feature ever conceived for a text editor. It should be noted that vi and Emacs fans fight bitter religious wars over which is better.	command line
nano	nano is a free clone of the text editor supplied with the pine email program. nano is very easy to use but is very short on features. I recommend nano for first-time users who need a command line editor.	command line
gedit	gedit is the editor supplied with the Gnome desktop environment.	graphical
kwrite	kwrite is the "advanced editor" supplied with KDE. It has syntax highlighting, a helpful feature for programmers and script writers.	graphical

VI Basics

Modes & Controls

Command Mode ESC (commands preceded by :)

Insertion Mode Entered on insertion or change

Starting VI (command line)

vi <filename> Edit *filename*

vi -r <filename> Edit last version of *filename* after crash

vi +n <filename> Edit *filename* at line *n*

vi + <filename> Edit *filename* at end of file

vi +/str <filename> Edit *filename* at first occurrence of *str*

In insertion mode the following should be preceded by ESC:

:w Save

:x Save & Exit

:q Exit if no changes made

:q! Exit & discard any changes

Show vi and nano

Hello World

- Open up the text editor of your choice and type. Save it as *hello_world*.

```
#!/bin/bash

# This is our first script.

echo 'Hello World! '
```

- Comments in the shell are made with “#”
- The first line is a special sequence
 - **#!** is called *shebang* and it tells the system what interpreter to use when executing the script.
 - Every shell script should include this
 - Have you ever seen a python script with **#! usr/bin/python** at the top?

Hello World

- We saw last week that default permissions are *rw-r--r--*, so we will need to use *chmod* to make it executable:
 - `$ chmod 700 hello_world` makes it executable for just the owner (you)
 - `$ chmod 755 hello_world` makes it executable for everyone
- We can execute the script with `$./hello_world`
 - Why can't we just type `$ hello_world`?
 - If the PATH variable doesn't contain the current working directory, then we need to provide the path to the script at the front.

Next Class