Adapted from Software Carpentry # Opening a Unix Shell in Bash

Where to type commands: How to open a new shell

The shell is a program that enables us to send commands to the computer and receive output. It is also referred to as the terminal or command line.

Some computers include a default Unix Shell program. The steps below describe some methods for identifying and opening a Unix Shell program if you already have one installed. There are also options for identifying and downloading a Unix Shell program, a Linux/UNIX emulator, or a program to access a Unix Shell on a server.

If none of the options below address your circumstances, try an online search for: Unix shell [your computer model] [your operating system].

Linux

The default Unix Shell for Linux operating systems is usually Bash. On most versions of Linux, it is accessible by running the (Gnome) Terminal or (KDE) Konsole or xterm, which can be found via the applications menu or the search bar. If your machine is set up to use something other than Bash, you can run it by opening a terminal and typing bash.

macOS

For a Mac computer running macOS Mojave or earlier releases, the default Unix Shell is Bash. For a Mac computer running macOS Catalina or later releases, the default Unix Shell is Zsh. Your default shell is available via the Terminal program within your Utilities folder.

To open Terminal, try one or both of the following:

In Finder, select the Go menu, then select Utilities. Locate Terminal in the Utilities folder and open it.

Use the Mac 'Spotlight' computer search function. Search for: Terminal and press Return.

To check if your machine is set up to use something other than Bash, type echo $SHELL in your terminal window.

If your machine is set up to use something other than Bash, you can run it by opening a terminal and typing bash.

Reference

How to Use Terminal on a Mac

Windows

Computers with Windows operating systems do not automatically have a Unix Shell program installed. In this lesson, we encourage you to use an emulator included in Git for Windows, which gives you access to both Bash shell commands

and Git. If you are attending a Software Carpentry workshop session, it is likely you have already received instructions on how to install Git for Windows.

Once installed, you can open a terminal by running the program Git Bash from the Windows start menu.

Other solutions are available for running Bash commands on Windows. There is now a Bash shell command-line tool available for Windows 10. Additionally, you can run Bash commands on a remote computer or server that already has a Unix Shell, from your Windows machine. This can usually be done through a Secure Shell (SSH) client. One such client available for free for Windows computers is PuTTY. See the reference below for information on installing and using PuTTY, using the Windows 10 command-line tool, or installing and using a Unix/Linux emulator.

Reference

Git for Windows - Recommended

For advanced users, you may choose one of the following alternatives:

Install the Windows Subsystem for Linux

Using a Unix/Linux emulator (Cygwin) or Secure Shell (SSH) client (Putty)

*Please note that commands in the Windows Subsystem for Linux (WSL) or Cygwin may differ slightly from those shown in the lesson or presented in the workshop.

Session 1: Shell for Navigating to Files and Directories

Your challenges of the day:

1. Does type case matter? Is there a difference between `ls -s` and `ls -S`?
2. Do spaces matter? Is there a difference between `ls-F` and `ls -F`?

Explore more `ls` flags.
1. What does `-l` option do? What if you use `-l` and `-h`? 1. The default `ls` lists contents in alphabetical order. What option do I use to see them by time of last change?

Questions of the day:

- What is a command shell and why should I use one?
- How can I move around on my computer?
- How can I see what files and directories I have?
- How can I specify the location of a file or directory on my computer?

What is Unix Shell?

- *It is different from how we usually interact with our devices, on a* ***graphical user interface*** *(GUI)*
- Shell is a **Command-Line Interface** (CLI)
- Type commands in the **prompt** `$`
- Invoke complicated programs
- Shell is a scripting language
- We will use the Unix Shell: Bash (Bourne Again SHell by Stephen Bourne)

Why use Bash?

- Combine existing tools into powerful pipelines and handle large volumes of data automatically.
- Sequences of commands can be written into a script, improving the reproducibility of workflows.
- Essential to interface with hardware, HPCC, and remote machines.
- Commands and the grammar of shell are used in other coding languages.

Navigating files and directories

- **File System**: The part of the operating system responsible for managing files and directories
- **Files** hold information
- **Directories** (or **folders**) hhold files or other directories. Thinks of them like *places*.
- **Current working directory** is the place where you are in the file system when you are using the shell.
- **Root directory** is the top directory that holds everything else. It is refered to by a slash `/` on its own. This is the leading slash in other directory paths, for example `/Users/claire/`
- **Hidden files and directories** start with `.` like `.bash_profile`. They are usually configuration settings and are hidden to prevent cluttering the terminal with a standard `ls` command. Add the `-a` option see hidden files.

• Commands of the day:

- `ls`: listing. This command will list the contents of the current directory
- `-F` option (switch or flag) tells ls to classify the output by adding a marker to file and directory names to indicate what they are.
- `-s` option displays the size of files and directories
- `-S` option will sort the files and directories by size
- `--help` option will tell us how to use the command and what options it accepts
- `pwd`: print working directory. Directories are like places, at any time while we are using shell, we are in one place, called our current working directory
- `clear`: clears the terminal if it gets to cluttered
- up and down arrows can be used to access previous commands (or scroll)
- `man` will give you the manual for a command, for example `man ls` will tell us all about listing
- `cd` will change your working directory. `cd` can only see sub-directories inside your current working directory.
- `cd ..` is a shortcut to move up one directory to the *parent directory* of the one we are in

- `cd ~/` is a shortcut to move to the current user's home directory. For example, if my home directory is `/Users/claire`, then `~/data` is equivalent to 'Users/claire/data'

General syntax of a shell command

Bash

```
$ ls -F /
```

`ls` is the **command**, with an **option** (or **switch** or **flag**) `-F` and an **argument** `/`. **Options** start with a single dash (`-`) or two dashes (`--`) and change the behavior of the command. Arguments tell the command what to operate on (e.g. files and directories). Options and argements are refered to as **parameters**.

Type case is important. Spaces are important between command and options. (But options can be combined with a single - and no spaces)

Getting help

- The help option can be used with a command, for example `ls --help`
- The manual (`man`) for a command can be accessed, for example `man ls`

References

- Intermediate Linux Commands
- Software Carpentry Unix Shell

Session 2: Working with files and directories

Questions of the day:

- How can I create, copy, and delete files and directories?
- How can I edit files?

Challenge Questions

1. Moving files. We accidentally put the files `sucrose.dat` and `maltose.dat` into the wrong folder, `analyzed/`. Fill in the blanks to move these files into the `raw/` folder.

   ```
   $ ls -F
     analyzed/  raw/
   $ ls -F analyzed/
     fructose.dat glucose.dat maltose.dat sucrose.dat
   $ cd analyzed
   ```

   My next line of code should be (fill in the blanks):

   ```
   $ mv sucrose.dat matose.dat ___/___
   ```

   Solution Think about ../raw Recall that .. refers to the parent directory (i.e. one above the current directory).

2. Renaming Files. We mispelled a filename! Which of the following commands will correct our mistake?

   - a. `cp statstics.txt statistics.txt`
   - b. `mv statstics.txt statistics.txt`
   - c. `mv statistics.txt .`
   - d. `cp statstics.txt .`

Solution

(a.) Will copy the file, so we will end up with the mispelled and correct version. (b.) Will move (i.e. rename) the incorrect file name to a correct filename. (c.) and (d.) will not work. Remember . is the current directory.

3. Removal. What happens when we execute `rm -i thesis/finaldraft.txt`? Why would we want this protection when using `rm`? Solution The program will confirm that we want to delete the thesis final draft file. Remember, deletion is forever! There is no trash can or recycle bin.

4. Removal. What is wrong with the command `rm -i thesis`? Solution The remove command will not act on a directory unless the recursive option (-r)is given.

5. Removal. What is wrong with the command `rm -r thesis`? Solution This remove command will delete the directory thesis and all its contents, but we forgot to check for confirmation with the interaction option (-i). Remember, deletion is permanent!

6. Wildcards. Which of the following matches the file names `ethane.dat` and `methane.dat`?

- a. `ls ?ethane.dat`
- b. `ls *ethane.dat`
- c. `ls ???ane.dat`
- d. `ls ethane.*` Hint Remember ? wildcard matches to exactly one character. * wildcard can match to zero to many characters.

Tips for good names for files and directories

1. Don't use spaces. Use `-` or `_` or *camelCase*.
2. Don't begin a name with a `-` (dash). It will look like a command option. Names should start with letters or numbers.
3. Avoid special characters. Some have special meanings.

If you need to refer to names of files or directories that have spaces, put them in quotes ("").

What's in a name?

A **filename extension** is the second part of the filename after the dot (.). They help us and programs tell different kinds of files apart. A few examples: - .txt: plain text file - .csv: comma separated value file - .pdf: PDF document - .cfg: configuration file of parameters for a program - .png: an image file

The **wildcard** `*` matches zero or more characters. For example, to access all the text files in a directory, use `*.txt`.

The **wildcard** `?` matches exactly one character.

Commands of the Day

- Creating Directories or Files:
- `mkdir path` creates a new directory
- `nano new` runs a text editor called Nano to create a new file by the new name given. For example, `nano thesis.txt` creates a text file named `thesis.txt` in the working directory.
- `touch new` creates an empty (0 byte) file by the new name given. Why bother? Some programs require empty files to populate with output.
- Moving or Renaming directories or files safely:
- `mv old new` command move has two arguments. The first tells `mv` what we're moving, while the second is where it's to go.
- `mv -i` or `mv -interactive` must be used to make `mv` ask for confirmation before overwriting any existing file or directory with the same name as the second argument. (Otherwise, Beware! It will silently overwrite.)
- Copying directories and/or files:

- `cp old new` command copies a file (first argument) to a new location (second argument)
- `cp -r` adds the recursive option to copy a directory and all its contents to another directory (second argument). For example, we can make a backup with `cp -r thesis thesis_backup`.
- `cp` can be used on multiple filenames as long as a destination directory is the last argument. For example, `cp a.txt b.txt c.txt backup/` will copy the three text files into the subdirectory `backup/`.
- Removing files and directories safely: **Deleting is forever**
- `rm -i path` command for remove with interactive option to ask for confirmation before deleting.
- `rm -i -r path` command with interactive option and recursive option will **remove a directory and all its contents** with confirmation prompts.

Which editor should I use?

- `nano` is a built-in text editor that only works with plain character data (i.e. no tables, images, or other media). It is the least complex, but you may want to try more powerful editors.

For Unix Systems (Linux and macOS) - Emacs - Vim - Gedit is a graphical editor

For Windows - Notepad++ - `notepad` is built-in and can be run in the command line

If you start an editor from the shell, it will use your current working directory as its default location.

In editor commands, the Control key is also called Ctrl or ^.

Challenge Project

Before heading on a trip, you want to back up your data and send some datasets to Claire. Fill in the following commands to get the job done. First, let's set up a directory and files.

```
# Hashtag denotes a comment. The line will be skipped

# Change to your desktop
cd ~/Desktop

# Make a new folder for our fake data
mkdir fake_data
cd fake_data

# Create some empty files.
touch 2020-06-09-data.txt
touch 2020-06-09-calibration.txt

# Make sure the new files are created. Notice we can combine options)
ls -Fs

# Let's add some info to our file and confirm it with the editor (spoiler alert - redirects!
echo Hello World > 2020-06-09-data.txt
nano 2020-06-09-data.txt

# Let's edit and add information to another.
nano 2020-06-09-calibration.txt
```

The next piece is provided in the shell script `session2challenge.sh`. Copy it to your fake_data directory.

```
# session2challenge.sh creates more fake data and callibration files

fmonth="2020-06"
echo $fmonth

# Loop through days to create data files and calibration files
for i in `seq -w 10 30`
do
  # Define the filename
  printf -v fname '%s-%02d-data.txt' "$fmonth" "$i"
```

```
  # Create an empty file
  touch "$fname"
  # Redirect in some data
  echo data $i > "$fname"

  printf -v fname '%s-%02d-calibration.txt' "$fmonth" "$i"
  touch "$fname"
  echo $i > "$fname"
done
```

Now, it's your turn! 1. Create a backup directory with separate subdirectories for data and calibration files. Copy files to the appropriate locations. 1. Create a directory named send_to_claire and copy all the data from June 11th to it.

Add your code to the Jamboard!

Get a hint

Create a backup directory with subdirectories for data and calibration files - Hint: You will use `mkdir` `mkdir ___` `mkdir ___/___` `mkdir ___/___`

Copy data files to `backup/data`. (Use a similiar approach for calibration files.) - Hint: Use the copy command `cp` with wildcards `cp *-data.txt backup/___`

Copy June 11th files to `send_to_claire/`. - Hint: Use the copy command `cp` with wildcards! `cp *-11-*.txt send_to_claire/`

Session 3: Pipes and Filters

The idea of linking together programs is why Unix has been so successful. Instead of creating enormous programs that try to do many different things, we focus on lots of simple tools that work well with each other.

Challenge Questions:

1. In our current directory, we want to find the three files which have the least number of lines. Which command listed below would work?

- a. `$ wc -l * > sort -n > head -n 3`
- b. `$ wc -l * | sort -n | head -n 1-3`
- c. `$ wc -l * | head -n 3 | sort -n`
- d. `$ wc -l * | sort -n | head -n 3`

2. See the file called `data-shell/data/animals.txt`. What text passes through each of the pipes and the final redirect in the pipeline below?

```
$ cat animals.txt | head -n 5 | tail -n 3 | sort -r > final.txt
```

Hint: Build the pipeline up one command at a time to test your understanding.

3. `uniq` filters out adjecent matching lines in a file.
   How can we extend the pipeline to find out what animals the file `data-shell/data/animals.txt` contains without any duplicates? Solution cut -d , -f 2 animals.txt | sort | uniq > animals_unique.txt

4. Assuming your current working directory is `data-shell/data/`, which command would you use to produce a table that shows the total count of each type of animal in the file `animals.txt`?

- a. `$ sort animals.txt | uniq -c`
- b. `$ sort -t, -k2, 2 animals.txt | uniq -c`
- c. `$ cut -d, -f 2 animals.txt | uniq -c`
- d. `$ cut -d, -f 2 animals.txt | sort | uniq -c`
- e. `$ cut -d, -f 2 animals.txt | sort | uniq -c | wc -l`

Questions of the day:

- How can I combine existing commands to do new things?
- How can I write to a file from the shell prompt?

Commands We Already Know

- Navigating File System
- `ls`: listing contents of working directory with many options: `-F` classify, `-a` list all, `-s` size, `-S` sort by size
- `pwd` print working directory
- `clear` the terminal
- `man` will give you the manual for a command
- `cd` will change working directory
- `cd ..` change up to parent directory
- `cd ~` change to home directory
- Creating Directories or Files:
- `mkdir path` creates a new directory
- `nano new` runs a text editor
- `touch new` creates an empty (0 byte) file
- Moving or Renaming directories or files safely:
- `mv -i old new`
- Copying directories and/or files:

- `cp old new`
- `cp -r` to copy a directory and all contents
- Removing / Deleting Safely: **Deleting is forever**
- `rm -i path` delete file with confirmation
- `rm -i -r path` delete directory and contents

- Wildcards
- `?` matches to one character
- `*` matches to zero to many characters

Commands of the Day

- **Filters** are programs that transform a stream of input into a stream of output
- `wc` is the word count command for number of lines, words, and characters in a file (left to right in that order)
- `echo` prints a string or the value of a variable as output. For example 'echo The echo commmand prints text`or`echo $SHELL$`printsthevalueofthevariable`SHELL`

(a defined path)

- `sort` sorts the contents of a file. `sort -n` sorts a numerical file.
- To escape a mistake in the prompt, type [Control] + [C]
- Write to a file from the prompt
- `>` **redirects** a command's output to a file instead of printing it to the screen. DO NOT write to the same file.
- `>>` **redirects* a command's output to append to the end of a file
- View particular file contents
- `cat`is the concatentate (join together) command that prints the contents of files one after another
- `less` displays a screenful of the file and then stops. You can go forward one screenful by pressingthe spacebar, or back one by pressing [b] and [q] to quit.
- `head` shows the first few lines of a file. For example, `head -n 5` will show the first 5 lines.
- `tail` shows the last few lines of a file
- `cut` removes or cuts out certain sections of each line in a file
  - `-d` option specifies a delimeter
  - `-f` option specifies the column for extraction
- `uniq` filters out adjecent matching lines in a file.
- Piping Commands Together
- `|` command **pipe** tells the shell to use the output of a command on the left as the input of the command on the right
- Chain pipes consecutively

Session 4: Loops

Linking together programs is why Unix has been so successful. Now, we improve productivity through automation – with loops! All those commands we have learned will be put to use.

Nested Loops Challenge!

What does this do?

```
$ for species in cubane ethane methane
> do
>     for temperature in 25 30 37 40
>     do
>         mkdir $species-$temperature
>     done
> done
```

Questions of the day:

- How can I perform the same actions on many different files?

Commands of the Day

Loop Structure `{Loop Structure} for thing in list_of_things do operation_using $thing done`

Loop Examples

List the contents of working directory one item at a time

```
for itemname in *
do
   ls $itemname
done
```

Output part of files in a directory

```
cd Desktop/data-shell/creatures
for filename in basilisk.dat minotaur.dat unicorn.dat
do
   head -n 2 $filename
done
```

Write files in a directory to a new file

```
cd Desktop/data-shell/creatures
for filename in basilisk.dat minotaur.dat unicorn.dat
do
   cat -n 2 $filename >> all.pdb
done
```

Repeat running a program with all your input data files

Nell has files NENE00000A.txt and NENE00000B.txt that need needs to run through th program `goostats` one at a time. The program `goostats` has two arguments, the input data file, and the output statistics file.

```
cd ../north-pacific-gyre/2012-07-03
for datafile in NENE*[AB].txt
do
    echo $datafile
    bash goostats $datafile stats-$datafile   #where stats-$datafile is the output of goosta
done
```

Checking on your loop before you run it!

It can be a good idea to run your loop with `echo` in front of you commands, to make sure it will act the way you believe. For example, in the loop above I may want to first run `echo "bash goostats $datafile stats-$datafile"` before I run the loop to execute the `goostats` program.

Commands We Already Know

- Navigating File System
- `ls`: listing contents of working directory with many options: `-F` classify, `-a` list all, `-s` size, `-S` sort by size
- `pwd` print working directory
- `clear` the terminal
- `man` will give you the manual for a command
- `cd` will change working directory
- `cd ..` change up to parent directory
- `cd ~` change to home directory
- Creating Directories or Files:
- `mkdir path` creates a new directory
- `nano new` runs a text editor

- `touch new` creates an empty (0 byte) file
- Moving or Renaming directories or files safely:
- `mv -i old new`
- Copying directories and/or files:

- `cp old new`
- `cp -r` to copy a directory and all contents
- Removing / Deleting Safely: **Deleting is forever**
- `rm -i path` delete file with confirmation
- `rm -i -r path` delete directory and contents

- Wildcards
- `?` matches to one character
- `*` matches to zero to many characters
- Filters
- `wc` is the word count
- `echo` prints text or the value of a variable
- `sort` sorts the contents of a file. `sort -n` sorts numerically.
- Write to a file from Prompt
- `>` **redirects** a command's output to a file
- `>>` **redirects* a command's output to append to end of a file
- View particular file contents
- `cat`concatentate prints the contents of files
- `less` displays a screenful of the file and then stops
- `head` shows the first few lines of a file
- `tail` shows the last few lines of a file
- `cut` removes or cuts out certain sections of each line in a file
  - `-d` option specifies a delimeter
  - `-f` option specifies the column for extraction
- `uniq` filters out adjecent matching lines in a file.
- Piping Commands Together
- `|` command **pipe** tells the shell to use the output of a command on the left as the input of the command on the right

Session 5: Shell Scripts

We finally see what makes the shell a powerful programming environment.
We will take commands we repeat and save them in a **shell script**- a small program, so we can re-run operations with a single command.

Questions of the day:

- How can I save and re-use commands?

Fun Resources

- Bash Help Sheet has shortcuts for quick navigating and editing in your shell
- Mastering Bash with Tips and Tricks has some great examples of how scripts can be used in a variety of ways.v
- 30 Bash Script Examples depicts some basic to more complex scripting examples
- StackOverflow thread of most powerful examples of Unix Commands or Scripts every programmer should know is old but has some great examples. In general, StackOverflow is a great community for technical questions.

Writing Shell Scripts

Shabang the top line of a script:

```
#!/bin/bash
```

Uses the special marker `#!` and path `/bin/bash` the instruct the shell to pass the script to the bash program for execution.
Other scripts may point to other shells (e.g. `#!/usr/bin/perl` will tell the shell to run a perl script.)

Use an argument on the command line executing a script

For example, `$1` means the first argument on the command line in the script `header.sh`.

header.sh:

```
#!/bin/bash
# This script prints the first 15 lines of the file named in the command line (datafile.txt)
head -n 15 $1
```

Command line:

```
$  bash header.sh datafile.txt
```

Use multiple arguments on the command line executing a script

- Use double quotes around a variable in case a filename happens to contain spaces.
- Use special variables `$1`, `$2`, and `$3`, etc. **header.sh:**

  ```
  #!/bin/bash
  # This script prints the top $2 lines of the file $1, then writes the top lines to file
  head -n "$2" "$1" > "$3"
  ```

  Command line:

  ```
  $  bash header.sh datafile.txt 10 topdata.txt
  ```

Use special syntax to handle one or more filenames

- Use `$@` to indicate all of the command-line arguments to the shell script. Add quotations in case of filename spaces `"$@"` **sorted.sh:**

  ```
  #!/bin/bash
  # Sort files by their length
  # USAGE: bash sorted.sh one_or_more_filenames
  wc -l "$@" | sort -n
  ```

  Command line:

  ```
  $  bash sorted.sh  *.pdb  ../creatures/*.dat
  ```

Commands and Concepts We Already Know

- Navigating File System
- `ls`: listing contents of working directory with many options: `-F` classify, `-a` list all, `-s` size, `-S` sort by size
- `pwd` print working directory
- `clear` the terminal

- `man` will give you the manual for a command
- `cd` will change working directory
- `cd ..` change up to parent directory
- `cd ~` change to home directory
- Creating Directories or Files:
- `mkdir path` creates a new directory
- `nano new` runs a text editor
- `touch new` creates an empty (0 byte) file
- Moving or Renaming directories or files safely:
- `mv -i old new`
- Copying directories and/or files:

- `cp old new`
- `cp -r` to copy a directory and all contents
- Removing / Deleting Safely: **Deleting is forever**
- `rm -i path` delete file with confirmation
- `rm -i -r path` delete directory and contents

- Wildcards
- `?` matches to one character
- `*` matches to zero to many characters
- Filters
- `wc` is the word count
- `echo` prints text or the value of a variable
- `sort` sorts the contents of a file. `sort -n` sorts numerically.
- Write to a file from Prompt
- `>` **redirects** a command's output to a file
- `>>` **redirects* a command's output to append to end of a file
- View particular file contents

- `cat`concatentate prints the contents of files
- `less` displays a screenful of the file and then stops
- `head` shows the first few lines of a file
- `tail` shows the last few lines of a file
- `cut` removes or cuts out certain sections of each line in a file
  - `-d` option specifies a delimeter
  - `-f` option specifies the column for extraction
- `uniq` filters out adjecent matching lines in a file.
- Piping Commands Together
- `|` command **pipe** tells the shell to use the output of a command on the left as the input of the command on the right
- Loop Structure:

```
for thing in list_of_things
do
   operation_using $thing
done
```

Session 6: Finding Things!

Questions of the day:

How can we find files? How can we find things in files?

Commands of the Day

- `grep` is a contraction of global/regular expression/print. It finds and prints lines in files that match a pattern.
- **regular expressions** are patterns that can include wildcards
- Usage: `grep pattern filename`
- `grep -w` limits to word boundaries
- `grep -n` prints the line numbers that match
- `grep -i` makes search case-insensitive
- `grep -v` inverts the search to output that does not contain the pattern
- `grep -E` notes that the pattern is an extended regular expression that can contain wildcards
- `find` command finds files!
- `-type` d or f for directories or files
- `-name` matches a name, but look out for order of execution! Filenames with wildcards need quotes. For example, `find . -name "*.txt"`
- `$()` to combine commands. Code inside this runs first!
- For example, `wc -l $(find . -name "*.txt")`

Commands and Concepts We Already Know

- Navigating File System
- `ls`: listing contents of working directory with many options: `-F` classify, `-a` list all, `-s` size, `-S` sort by size
- `pwd` print working directory
- `clear` the terminal
- `man` will give you the manual for a command

- `cd` will change working directory
- `cd ..` change up to parent directory
- `cd ~` change to home directory
- Creating Directories or Files:
- `mkdir path` creates a new directory
- `nano new` runs a text editor
- `touch new` creates an empty (0 byte) file
- Moving or Renaming directories or files safely:
- `mv -i old new`
- Copying directories and/or files:

- `cp old new`
- `cp -r` to copy a directory and all contents
- Removing / Deleting Safely: **Deleting is forever**
- `rm -i path` delete file with confirmation
- `rm -i -r path` delete directory and contents

- Wildcards
- `?` matches to one character
- `*` matches to zero to many characters
- Filters
- `wc` is the word count
- `echo` prints text or the value of a variable
- `sort` sorts the contents of a file. `sort -n` sorts numerically.
- Write to a file from Prompt
- `>` **redirects** a command's output to a file
- `>>` **redirects* a command's output to append to end of a file
- View particular file contents
- `cat`concatentate prints the contents of files

- `less` displays a screenful of the file and then stops
- `head` shows the first few lines of a file
- `tail` shows the last few lines of a file
- `cut` removes or cuts out certain sections of each line in a file
  - `-d` option specifies a delimeter
  - `-f` option specifies the column for extraction
- `uniq` filters out adjecent matching lines in a file.
- Piping Commands Together
- `|` command **pipe** tells the shell to use the output of a command on the left as the input of the command on the right
- Loop Structure:

```
for thing in list_of_things
do
   operation_using $thing
done
```

- Writing Shell Scripts (See Session 5 notes)