# Command Line - Part 1

## STAT 133

Gaston Sanchez

`github.com/ucb-stat133/stat133-fall-2016`

# GUIs

# Graphical User Interfaces

- Windows and Mac use a Graphical User Interface (GUI) for you to interact with the OS.
- GUIs are easy to learn
- GUIs rely on visual displays
- GUIs can be extremely useful
- GUIs have improved the friendliness and usability of computers

# GUIs or Command Line?

- However, GUIs come with trade-offs
- They don't allow you to have more control over what your computer can do
- Some operations are labor intensive and repetitive
- You organize things by clicking and dragging with the cursor (which reduces reproducibility)

# GUI Disadvantages

- Lack of repeatability
- Lack of reproducibility
- Some tasks may be labor intensive using a GUI
- GUIs limit analyses on a cluster of computers

# Command Line

# Command Line

- Instead of using a GUI, we can use a command line program
- The command line program is known as the **shell**
- By typing commands we perform tasks on the computer (without using a mouse)

# Shell

- You're working with a program called the **shell**
- The shell interprets the commands you enter
- It runs the program you've asked for
- It coordinates what happens between you and the operating system
- There are various kinds or flavors of shells: e.g. Bourne (BASH), Korn, C shell

# Command Line

- To interact with the shell we need a **terminal emulator**
- In Unix-like systems (e.g. Mac) the terminal is usually known as "terminal"
- Windows does not really provide a terminal; instead it provides the *command prompt*

# Command Prompt in Windows

## Finding MS Windows command prompt

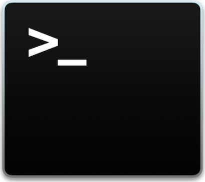
- Click the **Start** button
- Click **All Programs**
- Click **Accessories**
- Click **Command Prompt**

Windows command prompt is not a UNIX shell

# Shells for Windows

- Instead of using the command prompt you can use ad-hoc shell environments for Windows
- e.g. Git-Bash, PowerShell, Cygwin
- Git for Windows provides a BASH emulation
- PowerShell is part of Windows Management Framework 4.0
- Cygwin is large collection of GNU and Open Source tools

# Mac Terminal



- Go to **Applications**
- Go to **Utilities**
- Click **Terminal**

# Try Some Commands

- `date` (current time and date)
- `cal` (calendar of current month)
- `df` (amount of free space in your disk drives)
- `who` (logged in users)
- `echo ‘Hello’`

# Shell

- Shells run in terminal emulators, or **terminals**
- In Mac OS X, the default reminal program is called **Terminal**
- The command line is displayed within the terminal window
- The program behind the terminal is the **shell**
- There are many different shell programs

# BASH

The most common type of shell is BASH

- BASH: Bourne Again SHell
- BASH is the default shell for Linux
- BASH is usually the default shell on Mac
- type `echo $SHELL` to see your shell
- type `bash` to get a bash shell

# BASH

- A shell does much more than simply run commands
- It has wildcards for matching filenames
- It has a command history to recall previous commands quickly
- It has pipes for making the output of one command become the input of another
- It has variables for storing values for use by the shell

# Command `who`

- `who` displays a list of users that are currently logged in
- `who am i` (`whoami`) tells you the current user name

# Shell Command Syntax

```
command -options arg1 arg2
```

- Blanks and `"-"` are delimiters
- The number of arguments may vary
- An argument comes at the end of the command line
- It's usually the name of a file or some text
- Many commands have default arguments

# Date and Calendar

- `date`
- `cal` (current calendar year)
- `cal july 2015` (July 2015)
- `cal jan 2000`
- `ncal -w july 2015` (week number)

# Options

```
command -options arg1 arg2
```

- Options come between the command and the arguments
- They tell the command to do something other than its default
- They are usually prefaced with one or two hyphens
- e.g. `ncal -w july 2015`

# Some Control Sequences

| keys | description |
|---|---|
| `Ctrl + l` | clear screen |
| `Ctrl + c` | stop current command |
| `Ctrl + z` | suspend current command |
| `Ctrl + k` | kill to end of line |
| `Ctrl + r` | search history |
| `Ctrl + n` | next history item |
| `Ctrl + p` | previous history item |

# Manual Documentation

- To see the help documentation of a command use `man` followed by the name of the command:
  - `man cal`
  - `man date`
  - `man who`
- `q` quits manual documentation

# Logging Out

- `exit` logs you out
- `q` quits manual documentation

# System Navigation

# Filesystem Reminder

- The nested hierarchy of folders and files on your computer is called the **filesystem**
- The filesystem follows a tree-like structure
- The root directory is the most includive folder on the system
- The root directory serves as the container ofr all other files and folders
- A Unix-based system (e.g. OS X) has a single root directoyr
- Windows users usually have multiple roots (`C:`, `D:`, etc)

# Paths

- Each file and directory has a unique name in the filesystem
- Such unique name is called a **path**
- A path can be **absolute** or **relative**
- An **absolute path** is a complete and unambiguous description of where something is in relation to the root
- A **relative** describes where a folder or file is in relation to another folder

# Paths

- There are two special relative paths: `.` and `..`
- The single period `.` refers to your current directory
- The two periods means your parent directory, one level above

# Home Directory

- User's personal files are found in the `/Users` directory
- A user directory is the **home** directory
- `cd` (with no other arguments) returns you to your home directory
- `echo $HOME` prints your home directory
- `cd ~` takes you to your home directory

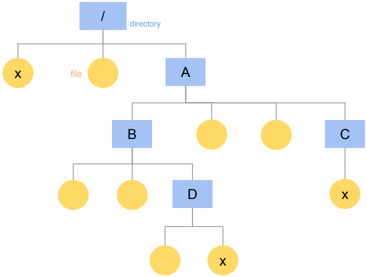
# Working Directory

- Another special type of directory is the so-called **working directory**
- The working directory is the current directory where you perform any task
- `pwd` prints the working directory

# Changing Directories

- `cd`
- `cd ..`
- `cd /`
- `cd ∼`
- `cd ∼/Documents`

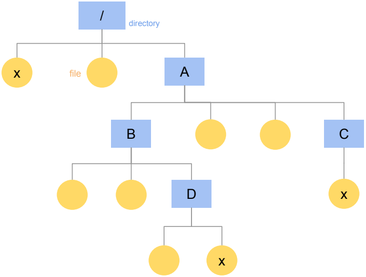
# Absolute Path Names



From the root directory to `D`:

```
cd /A/B/D
```

# Relative Path Names



Changing directories from `D` to `C`

```
cd ../../C
```

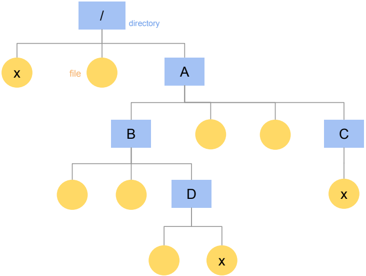
# Listing Contents in a Directory

- `ls`
- `ls -1` (one entry per line)
- `ls -l` (list in long format)
- `ls -a` (show files starting with a dot)
- `man ls` (manual documentation)

# Listing Contents in a Directory

- `ls /` (specify root directory)
- `ls /usr` (specifying a directory)
- `ls` $\sim$ (home directory)
- `ls -lt` (long format, sorted by modification time)

# Listing Contents



Show contents in D from C

```
ls ../B/D/
```

# Inspecting Files

# File Permissions

- run the command: `ls -l`
- directories may be displayed as: `drwxr-xr-x`
- files may be displayed as: `-rw-r--r--`
- file permissions are the 10 most left characters
- `r` means reads
- `w` means write
- `x` means execute

# File Permissions

Read from left to right the permissions mean

| position | description |
|---|---|
| 1 | File type. A dash – means a plain file and d means a directory. There are other less common options. |
| 2-4 | Owner permissions: read, write, and execute permissions for the file's owner. |
| 5-7 | Group permissions: read, write, and execute permissions for the file's group. |
| 8-10 | World permissions: read, write, and execute permissions for all other users. |

# Type of File

Determine the type of a file:
`file` *filename*

# Some commands for inspecting text files

- `wc` *filename*
- `cat` *filename*
- `head` *filename*
- `tail` *filename*
- `more` *filename*
- `less` *filename*

# Viewing file contents with `less`

- There are several commands that display the contents of text files
- The most commonly used file viewer is `less`
- `less` presents the contents of that file on the screen one page at a time
- There are various keyboard surtcuts to navigate in `less`

# Viewing file contents with `less`

| key | description |
|---|---|
| Page Up or **b** | scroll back one page |
| Page Down or **space** | scroll forward one page |
| Up Arrow | scroll up one line |
| Down Arrow | scroll down one line |
| **G** | move to the end of text file |
| **1G** or **g** | move to the beginning of the text file |
| **/hello** | search forward to next occurrence of hello |
| **n** | search for the next search occurrence |
| **h** | display help screen |
| **q** | quit less |

# Quoting Files

If you want a word to contain whitespace (e.g. a filename with a space in it), surround it with single or double quotes to make the shell treat it as a unit:

```
ls "My file"
```

# Exploring a file

- `cd` into a given directory
- List the directory contents with `ls -l`
- Determine the contents of a file with `file`
- If it looks like it might be text, try viewing it with `less`

# Editing text files at the command line

- Sometimes it is more convenient to create or modify a file right at the command line
- Although `less` is a convenient file viewer, it does not allow you to edit the contents
- Depending on your operating system and shell tool, you may have one or more command-line text editors:
- e.g. `vi`, `nano`, `gedit`

# Editing text files at the command line

- One common text editor is `vi` (there's also `vim`)
- It should be available in Mac, and also in Git-Bash (Windows)
- Depending on your operating system and shell tool, you may have one or more command-line text editors:
- Type `which vi` to fing out if you have it

# Editing text files with `vi`

- To create and start editing a file simply type `vi` followed by the name of the new file:

  ```
  vi newfile.txt
  ```

- Press the **I** key to start editing content
- When you're done, press the **ESC** key
- Then type `:wq` to save and quit
- You can reopen it again with: `vi newfile.txt`

Google **vi cheat sheet** to find more information

# File Management

# Managing Files

Common actions

- creating a directory
- creating a file
- copying a file
- moving a file
- deleting a file
- searching a file

# Managing Files

Common actions

- creating a directory: `mkdir`
- creating a file: usually through a text editor
- copying a file: `cp`
- moving a file: `mv`
- deleting a file: `rm`
- searching a file: ?

# Creating Directories and Files

Create a directory "summer2015" in my Documents

```
cd ~/Documents
mkdir summer2015
```

Create an empty file "README.md" in summer2015

```
cd summer2015
touch README.md
```

# Copying Files

- cp is the command to copy files
- cp can be used in two ways:
- `cp file1 file2` copies `file1` into `file2`
- `cp file1 directory` copies `file1` into a directory (directory must already exists)

# Copying Files

Copying `functions.R` from Documents to HW6

```
cp ~/Documents/functions.R ~/Desktop/HW6/
```

Copying `starwars.csv` to current directory

```
cp ~/Documents/starwars.csv .
```

# Deleting files

Deleting `README.md` and `starwars2.csv`

```
cd ~/Documents/summer2015
rm README.md
rm starwars2.csv
```

# Wildcards

- the shell provides special characters to specify filenames
- these special characters are called **wildcards**
- using wildcards allow you to select filenames based on patterns of characters
- these wildcards are similar to some regular expression characters

# Wildcards

| wildcard | description |
|---|---|
| `*` | matches any characters |
| `?` | matches any single character |
| `[characters]` | matches any character that is a member of the set *characetrs* |
| `[!characters]` | matches any character that is not a member of the set *characters* |
| `[[:class:]]` | matches any character that is a member of the specified *class* |

# Example

Create a directory `dummy`, `cd` to it, and then create empty files:

```
$ mkdir dummy
$ cd dummy
$ touch AGing.txt Bing.xt Gagging.text Going.nxt ing.ext
$ ls
```

# ∗ Wildcard

Use ∗ to refer to multiple files at once; it stands for *anything*

```
$ ls
  AGing.txt   Bing.xt
  Gagging.text Going.nxt ing.ext

$ ls G*
  Gagging.txt Going.nxt

$ ls *.xt
  Bing.xt
```

# ? Wildcard

The question mark ? represents a *single* character

```
$ ls
  AGing.txt   Bing.xt
  Gagging.text Going.nxt ing.ext

$ ls ?ing.xt
  Bing.xt
```

# [] Wildcard

Brackets [] can be replaced by whatever characters are within those characters:

```
$ ls
  AGing.txt   Bing.xt
  Gagging.text Going.nxt ing.ext

$ ls [B]ing.*
  Bing.xt

$ ls [A-G]ing.*
  Bing.xt
```

# Combining Wildcards

Wildcards can be combined:

```
$ ls
  AGing.txt   Bing.xt
  Gagging.text Going.nxt ing.ext

$ ls *G*
  AGing.txt Gagging.txt Going.nxt

$ ls *i*.*e*
  Gagging.text ing.ext
```

# Test Yourself

```
AGing.txt   Bing.xt
Gagging.text Going.nxt ing.ext
```

What command produces the output above:

A) `ls *ing.*xt`

B) `ls ?ing.*xt`

C) `ls ?ing.?xt`

D) `ls ?ing.xt`

E) `ls *ing.?xt`

# Test Yourself

```
AGing.txt Going.nxt ing.ext
```

What command produces the output above:

A) `ls *ing.*xt`

B) `ls ?ing.*xt`

C) `ls ?ing.?xt`

D) `ls ?ing.xt`

E) `ls *ing.?xt`

# Wildcard Examples

| Pattern | Matches |
|---|---|
| `*` | all files |
| `a*` | any file beginning with “a” |
| `*.txt` | any file ending with `.txt` |
| `b*.txt` | any file beginning with “b” followed by any characters and ending with “.txt” |
| `[gst]*` | any file beginning with either a “g”, and “s”, or a “t” |
| `[[:digit:]]*` | any file beginning with a number |
| `[[:upper:]]*` | any file beginning with an uppercase letter |

# Standard Input and Output

Many commands accept input
and produce output

# Input

Input can come from:

- the keyboard (a.k.a. **standard input**)
- other files
- other commands

# Output

Output can be:

- printed on screen
  - the command's results (a.k.a. **standard output**)
  - the status and error messages (a.k.a. **standard error**)
- written to files
- sent to other commands

# Output of commands

- Consider the command `ls`
- `ls` sends the results to a special file called: *standard output* or **stdout**
- `ls` sends status messages to another file called *standard error* or **stderr**
- By default both *stdout* and *stderr* are linked to the screen and not saved into a disk file

# SI and SO

- The “standard input” is usually your keyboard
- The “standard output” is usually your terminal (monitor)
- But we can also redirect inputs and outputs
- I/O redirection allows us to change where output goes and where input comes from
- I/O redirection is done via the > redirection operator

# Redirection Operator >

# The > operator

We can tell the shell to send the output of the `ls` command to the file `ls-output.txt`

```
ls -l ~/Documents > ls-output.txt
```

# The >> operator

We can tell the shell to send the output of the `ls` command and append it to the file `ls-output.txt`

```
ls -l ~/Desktop >> ls-output.txt
```

The contents in Desktop are appended to the file `ls-output.txt`

# Redirection

> redirects STDOUT to a file

< redirects STDIN from a file

>> redirects STDOUT to a file, but appends rather than overwrites

There is also << but its use is more advanced than what we'll cover

# About Redirection

- Many times it is useful to send the output of a program to a file rather than to the screen
- Redirecting output to files is very common when extracting and combining data (think of merge!)
- Think of the redirection operator ">" as an arrow that is pointing to where the output should go

# Joining files with cat

We can use cat and > to join two or more files:

```
# remember the files from HW5?
# (nflweather1960s.csv, ..., nflweather2010s.csv)
ls nflweather*s.csv

# joining all the decades files in one single file
cat nflweather*s.csv > allnfl.csv
```

The only issue here is that you would have appended column names

# Joining files with `cat`

Think about all the steps you would need to join the nfl-weather files without using the command line:

- You would have to open each file
- Open a new file `allnfl.csv`
- Start copy-pasting each adtaset into `allnfl.csv`
- Close all the decades files
- Save and close `allnfl.csv`

# Redirection with pipes

# Redirection

- The idea behind pipes is that rather than redirecting output to a file, we redirect it into another command
- STDOUT of one command is used as STDIN to another command
- We can redirect inputs and outputs
- Redirection is done via the | pipe operator

# Pipe example

Let's say you want to count the number of `.csv` files in a specfic directory:

```
# list csv files (one per line)
ls -1 *.csv

# piping to count lines with 'wc -l'
# (how many lines)
ls -1 *.csv | wc -l
```

The output of `ls -1` is piped to `wc -l`

# Pipe example

Let's say you want to inspect the contents of `/usr/bin`

```
# long list of contents
ls /usr/bin

# using 'less' as a pager to see all the contents
ls /usr/bin | less
```

The output of `ls` is piped to `less`

# Command `grep`

# Regular Expressions with `grep`

- We can work with some regular expressions in the command line
- For that purpose we use the command `grep`
- `grep` can be very helpful for extracting particular rows from a file

# grep example

Consider the data `nflweather.csv`

```
# rows containing Oakland (Raiders)
grep 'Oakland' nflweather.csv

# rows from 2013
grep '2013' nflweather.csv
```

# grep example

Consider the raw data `weather_20131231.csv`

```
# how many games in 2013
grep '2013' weather_20131231.csv | wc -l

# how many games in October 2013
grep '10/[0-9]*/2013' weather_20131231.csv | wc -l
```

# Command `curl`

# Command curl

- curl allows you to retrieve content from the Web
- curl stands for “see URL”
- It access Internet files on your behalf, downling the content without any need of a browser window

# curl example

```
# get the content of a URL
curl "http://www.stat.berkeley.edu/~nolan/data/stat133/Saratoga.txt"
```

# curl example

```
# get the content of a URL and save it to a file
curl "http://www.stat.berkeley.edu/~nolan/data/stat133/Saratoga.txt"
-o saratoga.txt


# equivalently
curl "http://www.stat.berkeley.edu/~nolan/data/stat133/Saratoga.txt"
> saratoga.txt
```

# Overview

# What good is it?

- Do I really need to learn these commands?
- The GUI file finder can do most of what we've seen (e.g. `ls`, `cd`, `mkdir`, `rmdir`)
- Maybe it can't do what `cut` can do, but so what?

# Advantages of shell commands

- Shell commands gives us a programatic way to work with files and processes
- They allow you to **record** what you did
- They allow you to repeat it another time
- Volumne: Have many many operations to perform
- Speed: need to perform things quickly
- Less error prone: want to reduce mistakes

# Command `cut`

# Command cut

- cut is most often used to extract columns of data from a field-delimited file
- They allow you to **record** what you did
- They allow you to repeat it another time

# cut example

```
# 2nd column of a tab-separated file
cut -f 2 starwarstoy.tsv


# 2nd column of a comma-separated file
cut -f 2 -d "," starwarstoy.csv
```

# cut example

```
# columns 2-4 of a tab-separated file
cut -f 2-4 starwarstoy.tsv


# columns 4-6 of a comma-separated file
cut -f 4-6 -d "," starwarstoy.csv
```

# cut example

```
# columns 2-3 of first 10 rows in nflweather
head -n 10 nflweather.csv | cut -f 2-4
```