Command Line - Part 2

STAT 133

Gaston Sanchez

Department of Statistics, UC–Berkeley

gastonsanchez.com
github.com/gastonstat
Course web: gastonsanchez.com/stat133

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
```