Data Analysis on the Command Line

Loosely based on the Jeroen Janssens Book
http://jeroenjanssens.com

Daniel P Cuneo

May 19, 2016

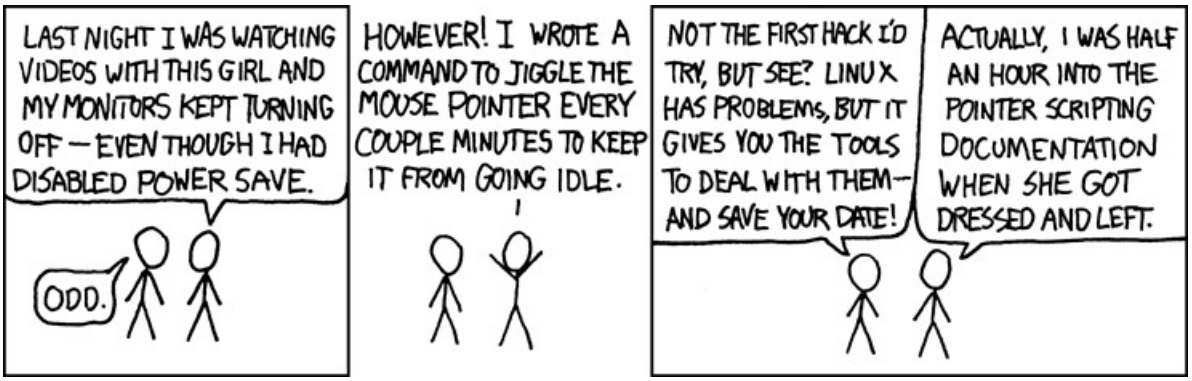

Figure 1: xkcd comic http://imgs.xkcd.com/comics/command_line_fu.png

1 Command Line Tools for Small Data

IPython run in a terminal is about the best way to work with data interactively. Loading a csv into a Pandas dataframe or just a Numpy array is very powerful. However, there are times when a handful of shell programs can either save the day or just make the day more productive.

Primarliy, I've found the program `find` to be extremely helpful, and combing it with `grep` or `pcregrep` can be enough to get a status report out.

I also loading data into Sqlite as the size of the test data grows. This has the advantage of being a easier to informally share with coworkers.

Csvkit was brought to my attention by the O'Reilly book, Data Science on the Command Line and I've come to really like it for small scale work. Note that Pandas can also push and pull data into and out of data base tables.

There are times when keeping my preliminary preprocessing steps completely shell driven is convenient especially if used with a makefile. I've been exploring the use of Drake, since it can encorporate native Python and I think that will be the future. Since GNU Make has been around a long time and is typically installed on any machine you're likely to come across, I've been trying to use it more and appreciate it before making the switch to Drake.

1.1 Version Information

```
In [1]: # https://github.com/rasbt/watermark
%load_ext watermark
%watermark -g -p numpy,scipy,pandas,matplotlib,statsmodels
```

```
numpy 1.9.3
scipy 0.16.0
pandas 0.16.2
matplotlib 1.4.3
statsmodels 0.6.1
Git hash: 22a1442ea56894a00796328d7be321ef8dfcf28d
```

1.2 Programs you may want to install

- `csvkit` by onryx, install with pip or conda
- `pcregrep` Debian sudo apt-get install
- `parallel` in the Debian repo.

1.3 Other items

I do a little contrast using Python and Pandas. You can skip running it if you don't already have it installed.

- mySQL
- Pandas

1.4 Links

- regular expression: http://www.rexegg.com/
- freeing memory http://unix.stackexchange.com/questions/87908/how-do-you-empty-the-buffers-and-cache-c
- `csvkit` https://csvkit.readthedocs.org/en/0.9.1/
- `find` command examples http://www.binarytides.com/linux-find-command-examples
- stackoverflow on excluding directories with `find` http://stackoverflow.com/questions/1489277/how-to-use-prune-option-of-find-in-sh
- fun site for inspiration http://www.commandlinefu.com/commands/browse

2 How to use this notebook

The ipython notebook is great for Python and there's some support for BASH and SQL. However, it's not perfect yet, and you will have to modify some lines.

The cells using bash commands, have the bash magic at the top and a shell variable:

```
%%bash

root="/home/daniel/git/Python2.7/DataScience/command\_line\_pres\_data"
```

That path is specific to my personal computer and will not work for you. You will need to change **every** occurance to the root path you downloaded the data set to.

In the future I'll consider using this BASH kernel for ipython notebooks:https://github.com/takluyver/bash_kernel. This kernel would allow me to export any shell variables once.

I also like this SQL magic provided here: https://github.com/catherinedevlin/ipython-sql

It prints SQL tables nicely and could be good for work reports based on SQL.

This notebook doesn't explain the individual commands in great detail. I am assuming that you either know them, or will be researching them on your own as you work through the commands. I have made some effort to make limited notes about the commands you see. Mainly, these notes are for explaining something not easily understood right away although I still assume you have googled or man paged the command.

3 Man Page

If you are very new to the command line you might not know how the man pages work. The manual is opened in vi like environment so navigation might be difficult at first. Pressing *h* while in the man pages will bring up a help file.

Here is a summary of a few commands for the man page viewer: * q to quit * / to search * n for next forward match * N for match backwards * h man page navigation help * j scroll up * k scroll down

4 Things I Don't Cover

4.1 Quoting

Quoting is tricky when dealing with the shell. The shell has to interpret (or not) everything passed in the command line. That means if there's spaces, or special tokens in the strings you are passing around, you'll have to escape the strings and wrap them in quotes, double or single.

This is a topic in itself. I didn't want to get sidetracked here so please read about it. In fact you'll have to because you'll end up banging your head against a wall sooner than later because of it.

A nice work around, is to write intermediate steps of a shell script to a file. Then read back the file. Since you are not passing the strings through the shell interpreter directly through stdin, they will not need special escaping or treatment. There's nothing wrong with this, it can help make things more clear to others, rather than using obscure escape sequences.

I made a point to use only *nice* file names here. Windoze people have a nasty habit of using spaces which is a huge PITA.

4.1.1 In brief:

1. Single quotes, ' , are for string literals with no special actions due to tokens
2. Double quotes, " , allow the expansion of shell variables and parameters, $FOO, and \ escape charaters
3. back ticks, ` , are completely different and are used to evaluate a command. Although I prefer the $(cmd) syntax

4.2 Regular Expressions

I used a handful of beefy regex's here that I chose because they cover a lot of ground. You'll want to spend a few weekends on the regegg.com site. I provide a quick overview of what the regex means in a cell but you'll have to look them up yourself.

https://en.wikiquote.org/wiki/Jamie_Zawinski
"Some people, when confronted with a problem, think"I know, I'll use regular expressions." Now they have two problems."

5 Imaginary use case

I'm given a hard drive full of source code and told that we will be doing a dependency analysis for some particular C libraries. We don't know in advance what we are looking for at the moment. That may be revealed on a client call or more information will be passed down. For now, we have the directories what can we do with it quickly ?

For this exercise I'm using the Linux Kernel 2.6 because it's sufficiently large and free to download.

The first thing I'd do, is inventory this thing we are given and prepare some basic report about what we have been given.

5.1 For this exercise we need an inventory

Below is is a simple inventory shell script, where we get the lines of code (including blanks, comments everything... not a real good method) and the file extension. This is more advanced than I plan to cover in this notebook.

This code goes into a file named run_inventory.sh which is included in the git repo if you cloned it.

In [25]:
```
%%bash # this line is here for syntax highlighting only

#!/bin/bash

function count_lines
{
    input=$1
    echo $(cat $input | wc -l)
}

function check_for_ascii
{
    input=$1
    bool=$(file $input | grep -ic "ascii")
    if [[ $bool -gt 0 ]];then
        echo 1
    else
        echo 0
    fi
}

function get_extension
{
    input="$1"
    base=$(basename "$input")
    test_=$(echo "$base" | grep -c "\.")

    if [ $test_ -eq 0 ];then
        echo "NONE"
    else
        ext=$(echo "$base" | rev | cut -d. -f 1 | rev)
    fi

    echo ${ext}
```

```
}

#### calls here ###
input=$1

count=$(count_lines $input)
ascii_bool=$(check_for_ascii $input)
ext=$(get_extension $input)

if [ $ascii_bool == 1 ];then
    printf "\"${input}\",\"${count}\",\"${ext}\"\n"
else
    printf "\"{$input}\",\"0\",\"${ext}\"\n"
fi
```

6 Make an inventory

The `find` program is tricky to learn, it has many options. I plan to dive more into `find`, but for now, I really just want an inventory.csv to play with.

I give `find` the path to search, which is our Linux Kernel directory, then I tell `find`, to only return the type, "f" which means files. No directories.

I then use a pipe '|' to pass the output into the input of the next program 'run_inventory.sh'. We need to use *xargs* to limit the way the output is presented to 'run_inventory.sh'. *Xargs* is another topic to learn about, so for now, just trust me.

In [5]:
```
%%bash
root="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"
#make a header
printf "\"path\",\"nlines\",\"ext\"\n" > $root/linux_inventory.csv
find $root/linux-2.6.32.67 -type f | xargs -n 1 $root/make_inventory.sh >> $root/li
```

The > is for stream redirection. The output stream from *printf* is sent to a new file. This will clobber existing files so practice with it. The second use is slightly different, >> is an appending stream redirection.

Since we don't want to clobber the linux_inventory.csv, we use the append operation for the actual data.

It's useful to be able to drop the header and add it back. You could do it in an editor, but since this notebook needs to be self contained, I add these cells. plus you can see another use for *sed* .

In [15]:
```
%%bash
#drop header
root="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

sed -i '1 d' $root/linux_inventory.csv
```

In [30]:
```
%%bash
# add header

root="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

header="\"path\",\"nlines\",\"ext\""
sed -i "1 i ${header}" $root/linux_inventory.csv
```

In [6]:
```
%%bash
root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"
```

```
        head -n 5 "${root_dir}/linux_inventory.csv"
```

```
"path","nlines","ext"
command_line_pres_data/linux-2.6.32.67/net/wireless/scan.c","1027","c"
command_line_pres_data/linux-2.6.32.67/net/wireless/core.h","401","h"
command_line_pres_data/linux-2.6.32.67/net/wireless/ibss.c","509","c"
command_line_pres_data/linux-2.6.32.67/net/wireless/nl80211.c","4896","c"
```

A neat trick to view the csv contents better. Works great in a real terminal, the ipython notebook's default cell width is too small to look nice :| So I hacked off part of the path output.
https://www.reddit.com/r/IPython/comments/27zash/can_i_increase_notebook_cell_width_on_wide_screens/

```
In [15]: %%bash
         root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

         head -n 10 "${root_dir}/linux_inventory.csv" | cut -d/ -f 5- | column -t -s,
```

```
"path"
Python2.7/DataScience/command_line_pres_data/linux-2.6.32.67/net/wireless/scan.c"
Python2.7/DataScience/command_line_pres_data/linux-2.6.32.67/net/wireless/core.h"
Python2.7/DataScience/command_line_pres_data/linux-2.6.32.67/net/wireless/ibss.c"
Python2.7/DataScience/command_line_pres_data/linux-2.6.32.67/net/wireless/nl80211.c"
Python2.7/DataScience/command_line_pres_data/linux-2.6.32.67/net/wireless/lib80211_crypt_tkip
Python2.7/DataScience/command_line_pres_data/linux-2.6.32.67/net/wireless/util.c"
Python2.7/DataScience/command_line_pres_data/linux-2.6.32.67/net/wireless/mlme.c"
Python2.7/DataScience/command_line_pres_data/linux-2.6.32.67/net/wireless/reg.h"
Python2.7/DataScience/command_line_pres_data/linux-2.6.32.67/net/wireless/wext-sme.c"
```

The pipe, |, takes output from another process, and supplies it as input to the next program. You will get different output for the two cells below. The first cell tells *grep* to operate on the output stream of the *find* command. The second tells *grep* to operate on files whose path is given as the output from *find*.

If you don't know what or how *head* works, try the man page for it. Look up the option n .

```
In [20]: %%bash
         root="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

         find $root/linux-2.6.32.67 -type f | grep "2.6" | tail -n 10
```

```
command_line_pres_data/linux-2.6.32.67/virt/kvm/ioapic.h
command_line_pres_data/linux-2.6.32.67/virt/kvm/irq_comm.c
command_line_pres_data/linux-2.6.32.67/virt/kvm/iodev.h
command_line_pres_data/linux-2.6.32.67/virt/kvm/coalesced_mmio.h
command_line_pres_data/linux-2.6.32.67/virt/kvm/coalesced_mmio.c
command_line_pres_data/linux-2.6.32.67/virt/kvm/Kconfig
command_line_pres_data/linux-2.6.32.67/virt/kvm/iommu.c
command_line_pres_data/linux-2.6.32.67/virt/kvm/eventfd.c
command_line_pres_data/linux-2.6.32.67/README
command_line_pres_data/linux-2.6.32.67/COPYING
```

```
In [21]: %%bash
         root="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

         find $root/linux-2.6.32.67 -type f | xargs grep "2.6" | tail -n 10
```

```
command_line_pres_data/linux-2.6.32.67/README:   not incremental and must be applied to the
command_line_pres_data/linux-2.6.32.67/README:   example, if your base kernel is 2.6.12 and
command_line_pres_data/linux-2.6.32.67/README:   2.6.12.3 patch, you do not and indeed must
command_line_pres_data/linux-2.6.32.67/README:   2.6.12.1 and 2.6.12.2 patches. Similarly, i
command_line_pres_data/linux-2.6.32.67/README:   version 2.6.12.2 and want to jump to 2.6.12
command_line_pres_data/linux-2.6.32.67/README:   reverse the 2.6.12.2 patch (that is, patch
command_line_pres_data/linux-2.6.32.67/README:   the 2.6.12.3 patch.
command_line_pres_data/linux-2.6.32.67/README:   Compiling and running the 2.6.xx kernels re
command_line_pres_data/linux-2.6.32.67/README:     kernel source code:       /usr/src/linux
command_line_pres_data/linux-2.6.32.67/README:   cd /usr/src/linux-2.6.N
```

7 let's make a summary of the kinds of files we were given by extension.

The next three cells are illustrating the how one can use cat, cut, sort and uniq to get a preliminary output of this inventory.

I broke out the steps so that you can follow along. There maybe short cuts and other arguments I didn't use. Sometimes it doesn't matter if you do something the correct way, just that you get it done quickly and you are confident that you know what you did. This type of prototyping is great, because you can see what's happening.

7.0.1 The program cut is super helpful. We can parse many output streams with cut.

If the format of a steam in tablular, then awk maybe the best, but awk is a whole animal into itself and I think most days, I rely on cut and then step into ipython and use Pandas or just Numpy.

The -d, option is to set the delimiter and the -f 3 says to use the 3rd field.

In [22]:
```
%%bash
root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"
cat $root_dir/linux_inventory.csv | cut -d, -f 3 | tail -n 5
```

```
"NONE"
"c"
"c"
"NONE"
"NONE"
```

In [23]:
```
%%bash
root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"
cat "${root_dir}/linux_inventory.csv" | cut -d, -f 3 | sort  | tail -n  5
```

```
"y"
"y"
"y"
"y"
"ymfsb"
```

In [24]:
```
%%bash
root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"
cat "${root_dir}/linux_inventory.csv" | cut -d, -f 3 | sort | uniq -c | tail -n 5
```

```
  1 "x86"
  105 "xml"
  6 "xsl"
  5 "y"
  1 "ymfsb"
```

```
In [32]: %%bash
         root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"
         cat $root_dir/linux_inventory.csv | cut -d, -f 3 | sort | uniq -ic | sort -n | ta
```

```
   857 "txt"
   1080 "S"
   2818 "NONE"
  11638 "h"
  13154 "c"
```

7.1 Putting this command line data into a database.

- csvkit: csvsql
- Pandas: df.to_sql()

7.1.1 Both Pandas and csvkit use SqlAlchemy to handle the connection to the DB.

I'm not expert at SqlAlchemy, but learning the connection strings offer value immediately. That's the, "mysql://¡log-in¿:¡passwd¿@¡ip¿/¡db-name¿" I'm running mySQL locally, localhost has IP 127.0.0.1 First I'll make the database using the mysqladmin program.

```
In [21]: %%bash
         root="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

         mysqladmin --user=root --password=test create LinuxKernel
```

csvsql will put the data into a table named "inventory" or if no –tables argument is given, named after the input file. It also deduces the data types.

```
In [22]: %%bash
         root="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

         csvsql --db "mysql://root:test@127.0.0.1/LinuxKernel" --tables "inventory" --inse
```

7.1.2 Check that it worked

The notebook isn't using the alias the way I expected so I had to type out the full command with user and password. In normal practice you'd make an alias as in the comments.

```
In [20]: %%bash
         #alias mysql='mysql --user=root --password=test'
         #mysql -e "SELECT * FROM inventory LIMIT 5;" LinuxKernel

         mysql --user=root --password=test -e "SELECT * FROM inventory LIMIT 5;" LinuxKerne
```

7.2 CSV outputs from mySQL without stepping into the sql shell

sql2csv returns a query in csv format to the stdout which can then be redirected to a file.

```
In [1]: %%bash
        sql2csv --db mysql://root:test@127.0.0.1/LinuxKernel --query "select ext, count(ext
            tail -n 5 | column -t -s,
```

```
dts  115
txt  857
S    1080
h    11639
c    13154
```

7.2.1 The numbers checkout.

Lets explore awk a little. The easiest thing to make *awk* do (and perhaps the most useful), is to print columns or rows from a file stream. awk * FS field separator set to comma * $3 is the 3rd column

grep * -E regular expression * -c count occurance (grep operates per line, we use pcregrep for multiple line searches)

There is a way to do the whole regex and count in awk I just didn't want to get into it. I'm still learning awk myself, and I haven't decided on it's usefulness over other tools. If I'm already in a database then forget it... I mostly want to show use of grep and awk for column parsing (rather than cut).

```
In [33]: %%bash
         root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

         cat "${root_dir}/linux_inventory.csv"  | awk  'FS=","; {print $3}' | grep -cE "txt
```

```
1715
```

8 Practice using *find* and *grep*

```
    find "${root_dir}/linux-2.6.32.67" -maxdepth 2 -mindepth 1 -type f -iname "*.c" | \
    xargs -n 1 pcregrep -no "(?sim)[a-z]+\w*\(.*?\)" /dev/null | \
    sed 's/\s//g' | \
    tail -n 10
    \subsection{find \$\{root\_dir\}/linux-2.6.32.67" -maxdepth 2 -mindepth
1 -type f -iname
``*.c"}\label{find-rootux5fdirlinux-2.6.32.67--maxdepth-2--mindepth-1--type-f--iname-.c}
```

- path to search
- -maxdepth 2 dont', search past 2 directories deep
- -mindepth 1, search at least 1 directory deep (using this to save time and the notebook kept crashing)
- -type f, only look for files not directories or links or anything else
- -iname, case insensitive glob style name matching, just like ls uses

8.1 xargs

A neat helper program, that takes the output from another shell program, and parses it into discrete intput arguments for the next program in a pipe.

This will allow us to pass one line at a time from find, to grep. Some programs do not need xargs, as they are designed to take a stream of input. Not in this case however.

8.2 Pearl Compatiple Regular Expression Grep (pcregrep)

```
pcregrep -no "(?sim)[a-z]+\w*(.*?)" /dev/null
```

- s dot matches everything including newlines

  `\\n`

- `i`
  case insensitive
- `m`
  multiline

The (.{?) makes the greedy "." stop, after encountering a left parenthesis, escaped like this \(This page explains the "Lazy Trap" issue. The greedy match "jumps the fence" http://www.rexegg.com/regex-quantifiers.html#lazytrap

8.2.1 This is going to hunt for functions

That is, strings that match

```
(?sim){[}a-z{]}+\w*(.*?)
```

where:

- [a-z] a list of lower case letters
- +
  at least once, or more
- \w
  any alpha-numeric zero or more times
- \(literal left parenthesis
- greedy match until right paren

The

```
/dev/null
```

is a trick to make grep print the file path it's working on.

8.3 sed

used to remove any spaces:

```
sed 's/\\s//g'
```

- s substitute
- \\s
  regular expression for any kind of while space
- // replace with nothing. . . .easier to read if it was, sed 's/foo/bar/g' , replace 'foo' with 'bar'
- g globally, as many times as a match can be made

```
In [5]: %%bash
        root_dir="/home/daniel/git/Python2.7/DataScience/notebooks/command_line_pres_data"

        find "${root_dir}/linux-2.6.32.67" -maxdepth 2 -mindepth 1 -type f -iname "*.c" | x
```

```
command_line_pres_data/linux-2.6.32.67/fs/dcookies.c:332:mutex_lock(&dcookie_mutex)
command_line_pres_data/linux-2.6.32.67/fs/dcookies.c:334:list_del(&user->next)
command_line_pres_data/linux-2.6.32.67/fs/dcookies.c:335:kfree(user)
command_line_pres_data/linux-2.6.32.67/fs/dcookies.c:337:is_live()
command_line_pres_data/linux-2.6.32.67/fs/dcookies.c:338:dcookie_exit()
command_line_pres_data/linux-2.6.32.67/fs/dcookies.c:340:mutex_unlock(&dcookie_mutex)
command_line_pres_data/linux-2.6.32.67/fs/dcookies.c:343:EXPORT_SYMBOL_GPL(dcookie_register)
command_line_pres_data/linux-2.6.32.67/fs/dcookies.c:344:EXPORT_SYMBOL_GPL(dcookie_unregister
command_line_pres_data/linux-2.6.32.67/fs/dcookies.c:345:EXPORT_SYMBOL_GPL(get_dcookie)
command_line_pres_data/linux-2.6.32.67/fs/no-block.c:15:no_blkdev_open(structinode*inode,stru
```

8.3.1 This example used the regular expression syntax where as above, I used the "globbing" rules.

```
In [ ]: %%bash
        root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

        find "${root_dir}/linux-2.6.32.67" -maxdepth 2 -mindepth 1 -type f -regextype posi
            xargs -n 1 pcregrep  -no "(?sim)[a-z]+\w*\(.*?\)" /dev/null | sed 's/\s//g' | t
```

We could use the "or" operator (-o) to `find` as well as the regex.

```
In [ ]: %%bash
        root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"
        find "${root_dir}/linux-2.6.32.67" -maxdepth 2 -mindepth 1 -type f -iname "*.c" -o
```

9 An aside about loops vs the *parallel* program

9.1 While loops in bash

```
In [ ]: %%bash
        root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

        while read f;do
            file_=$(echo $f | cut -d, -f 1 | sed 's/\"//g')
            grep -Eo -m 1 "^#include <linux"  "${file_}" /dev/null
        done < "${root_dir}/linux_inventory.csv"
```

```
In [ ]: %%bash
        root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

        cat $root_dir/prep.sh
```

```
In [ ]: %%bash
        root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

        tail -n 10 "${root_dir}/linux_inventory.csv" | xargs -P 4 -n 1 $root_dir/prep.sh
```

9.2 Parallel

A powerful program that makes running shell programs on multiple cores trivial, is parallel . It also cleans up code and makes things a single line when used with just a single core.

```
In [ ]: %%bash
        # Free the cache to really test the timing
        # become root and run
        # free && sync && echo 3 > /proc/sys/vm/drop_caches && free

        root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"
        time find "${root_dir}/linux-2.6.32.67" -maxdepth 2 -mindepth 1 -type f -name "*.c"
            parallel --jobs 1 -n 1 'pcregrep -no "(?sm)if\s*\(.*?\)" /dev/null' | sed 's/\s
```

```
In [ ]: %%bash
        # become root and run
        # free && sync && echo 3 > /proc/sys/vm/drop_caches && free
        time find "${root_dir}/linux-2.6.32.67" -maxdepth 2 -mindepth 1 -type f -name "*.c"
            parallel --jobs 4 -n 1 'pcregrep -no "(?sm)if\s*\(.*?\)" /dev/null' | sed 's/\s
```

10 Extend our use case

10.1 Add categorical columns to our inventory database

Let's say that we want to add a categorical columns to our SQL table. This is a nice way to store information about a table without worrying about normalization. We are not DBA's trying to maintain strict schema here. We just need a fast and intuitive way to query our data.

I had some difficulty when I first started reading about categorical variables. They are writen and talked about in various contexts. For now, I'm thinking about a column in a database table, that contains a string, which describes the row.

In contrast, I could have had a boolean flag like column. For every file with an include statement that has "linux" in it, give it a 1 and the rest a 0. Then for another condition, I'd have to add a flag for it, "foo" with 1 and 0's and so on.

In this case, a categorical variable is a single column that makes it wasy to do "group by", and aggregrate across categories. So let's add a column for the name of the directory,in the include statement, "#include ¡linux/*¿"

```
In [7]: %%bash
        root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

        find $root_dir/linux-2.6.32.67 -type f -iname "*.c" -o -iname "*.h" | \
            parallel -n 1 --jobs 4 'grep -Po -m 1 "(#include\s*\Wlinux/)\K\w+"' /dev/null \
            > "${root_dir}/path_cat_list.txt"
```

Again make a header as above with *sed* .

```
In [11]: %%bash
         root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

         sed -i '1 i\path:category' "${root_dir}/path_cat_list.txt"
```

Sanity check

```
In [41]: %%bash
         root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

         head -n 3 $root_dir/path_cat_list.txt
```

```
path:category
/home/daniel/git/Python2.7/DataScience/command_line_pres_data/linux-2.6.32.67/net/wireless/s
/home/daniel/git/Python2.7/DataScience/command_line_pres_data/linux-2.6.32.67/net/wireless/c
```

```
In [18]: %%bash
         root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

         csvsql -d ":" --db "mysql://root:test@127.0.0.1/LinuxKernel" --table "categories"
```

```
In [17]: %%bash
         mysql -uroot -ptest -e "select * from categories limit 5;" LinuxKernel
```

```
path        category
command_line_pres_data/linux-2.6.32.67/net/wireless/scan.c        kernel
command_line_pres_data/linux-2.6.32.67/net/wireless/core.h        mutex
command_line_pres_data/linux-2.6.32.67/net/wireless/ibss.c        etherdevice
command_line_pres_data/linux-2.6.32.67/net/wireless/nl80211.c        if
command_line_pres_data/linux-2.6.32.67/net/wireless/lib80211_crypt_tkip.c        err
```

Merge the two tables so that the original inventory table has the categorical columns filled in. UPDATE inventory, categories SET inventory.cat = categories.category WHERE inventory.path = categories.path;

```
In [ ]: %%bash
        sql="update inventory, categories set inventory.cat = categories.category where inv

        mysql -uroot -ptest -e "$sql" LinuxKernel
```

Get some summary output.

```
In [46]: %%bash
         sql2csv --db "mysql://root:test@127.0.0.1/LinuxKernel" --query "select ext, count
             head -n 10 | column -t -s,
```

```
ext  count(ext)  cat     count(cat)
c    2930        module  2930
c    2124        kernel  2124
c    1271        init    1271
h    875         types   875
c    654         types   654
c    299         delay   299
c    282         errno   282
c    200         fs      200
c    185         sched   185

[Errno 32] Broken pipe
```

10.2 Do the same thing with Pandas

```
In [3]: import sqlalchemy
        import pandas as pd
        import numpy as np
        from os import path
        root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"
        engine = sqlalchemy.create_engine("mysql://root:test@127.0.0.1/LinuxKernel")
```

10.2.1 Instead of grepping we use the python regex in a function

```
In [4]: import re

        ob = re.compile("#include\s\Wlinux/(?P<name>\w+)\.h\W")

        def regex_check(filename):
            fname = path.join(root_dir, filename)
            f = open(fname, 'r')
            match = ob.search(f.read())
            f.close()

            if match:
                return match.group('name')

            else:
                return None
```

Make sure the regex captures the right stuff. It only gets the first instance, which is fine. That's all the grep version above did. Not perfect, but OK for a day 1 iteration of what's in the directory we were given.

```
In [26]: %%bash
         root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

         sed -n "12,15p" $root_dir/linux-2.6.32.67/drivers/rapidio/rio-access.c
```

```
#include <linux/rio.h>
#include <linux/module.h>
```

```
In [5]: print regex_check('linux-2.6.32.67/drivers/rapidio/rio-access.c')

rio

In [6]: src = path.join(root_dir, "linux_inventory.csv")
        df = pd.read_csv(src, quotechar='"', quoting=1)
        df['category'] = df.apply(lambda row: regex_check(row['path']) if row['ext'] == 'h'

In [7]: df.loc[0:10, :]
```

Out[7]:

	path	nlines	ext	category
0	/DataScience/command...	1027	c	kernel
1	/DataScience/command...	401	h	mutex
2	/DataScience/command...	509	c	etherdevice
3	/DataScience/command...	4896	c	if
4	/DataScience/command...	788	c	err
5	/DataScience/command...	717	c	bitops
6	/DataScience/command...	679	c	kernel
7	/DataScience/command...	55	h	None
8	/DataScience/command...	402	c	etherdevice
9	/DataScience/command...	113	c	device
10	/DataScience/command...	259	c	None

I don't use Pandas' pivot table enough...I really like this tutorial and decided to incorporate it here. Once you go through the trouble of coding up a group by you might as well do a pivot table:
http://pbpython.com/pandas-pivot-table-explained.html

```
In [22]: table = df.pivot_table(index=['ext', 'category'], \
                                aggfunc={'ext':len, 'category':len, 'nlines':np.sum}, \
                                values=['ext', 'category', 'nlines'], \
                                margins=True)

         table.sort("nlines")[-10:]

         #table = df.pivot_table(index='ext', aggfunc={'nlines':np.sum,'ext':np.count_nonze
         #table.sort("nlines")[-10:]
```

Out[22]:

		category	ext	nlines
ext	category			
c	sched	186	186	137675
	errno	283	283	147777
	fs	200	200	157690
h	types	881	881	191313
c	delay	305	305	288431
	types	655	655	325444
	init	1277	1277	618232
	kernel	2124	2124	1103586
	module	2941	2941	2237201
All		30485	30485	10973169

10.3 Dump data frame to mySQL

```
In [ ]: df.to_sql("inventory2", engine, flavor='mysql')

In [ ]: %%bash
        sql="select count(path) as cts, category from inventory2 group by category order by
        mysql -uroot -ptest -e "${sql}" LinuxKernel
```

```
In [ ]: %%bash
        root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

        sql2csv --db "mysql://root:test@127.0.0.1/LinuxKernel" --query "select count(path)
        > "${root_dir}/categories.csv"
```

10.4 Back to Command line

10.5 Save the output to a file for the future

We'll redirect the output from standard out (terminal display) to a file.

```
In [37]: %%bash
         root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

         cat $root_dir/linux_inventory.csv | cut -d, -f 3 | sort | uniq -c | sort -n > $roo
```

10.6 Makefile

Let's try using a makefile. We have 2 steps required to create the sorted list of extensions and their counts.

1. make an inventory
2. sort the inventory by extension

We also have 2 dependencies

1. Linux kernel source
2. inventory

There's one final output, the sorted list of counts by extension

The idea behind the makefile, is that if we change a dependency, then we want the target output steps to run again. If a file was added to the Linux kernel, then we need a new inventory and then a file extension count list. in the a real world usage I'd make some more effort to avoid running the entire inventory. Here, that's a bit overkill.

10.7 What is happening

Make keeps track of when a file or directory has been modified. If something was touched, then the recipe is invoked to handle that updated information. Make can make use of functions, shell paramters although it a slightly different form.

Makefile are typically named, "makefile", and are tab delimited. Make has to parse the makefile so there's some special syntax that is similar to but distinct from that of the shell.

```
In [ ]: %%bash
        root_dir="/home/daniel/git/Python2.7/DataScience/command_line_data"

        cat -n makefile
```

```
In [ ]: %%bash
        root_dir="/home/daniel/git/Python2.7/DataScience/command_line_data"
        cat -n $root_dir/makefile
```

10.7.1 Test it out

We can use the *touch* command to update the modifcation dates of a file or directory. There are at least 3 ways to see the modifcation dates of a file, the most common being to *stat*. incidentaly, *stat* is a really basic program that is called internally in many other shell programs.

In [ ]:
```
%%bash
root_dir="/home/daniel/git/Python2.7/DataScience/command_line_data"

stat $root_dir/linux-2.6.32.67
```

The program *touch* is used to update the modification time to the present date. It is also used to create an empty file when one needs such a thing.

In [ ]:
```
%%bash
root_dir="/home/daniel/git/Python2.7/DataScience/command_line_data"

touch $root_dir/linux-2.6.32.67
stat $root_dir/linux-2.6.32.67
```

In [ ]:
```
%%bash
root_dir="/home/daniel/git/Python2.7/DataScience/command_line_data"
# if we are in the directory where the makefile exists, just issue the _make_ comma
# because this notebook could theoretically be run from anywhere, I'm using the ful
make -f $root_dir/makefile
```

In [ ]:
```
%%bash
root_dir="/home/daniel/git/Python2.7/DataScience/command_line_data"

stat $root_dir/ext_list.txt
```

10.8 Storing the output of command to a variable (shell paramter)

We can set a shell paramter from the output of another shell command/program. You've seen this trick used in the simple inventory program earlier. I used it a lot actually, to assign the output of commands to a shell paramter.

In [ ]:
```
%%bash

echo $SHELL # shell variable setup when you login
var=$(echo $SHELL | cut -d/ -f 2)
echo $var
```

Another example with the *date* program.

In [ ]:
```
%%bash
date
```

In [ ]:
```
%%bash
root_dir="/home/daniel/git/Python2.7/DataScience/command_line_data"

dir_path=${root_dir}/$(date +%Y_%m_%d_%H:%M:%S) # date can take a format string
echo $dir_path
```

10.9 Shell Loops

There are two kinds of loops that I tend to use:

1. while
2. for

10.9.1 Tests

The square brackets are called "tests" . This is another topic in shell scripting that I can't really cover right here but you can see a use case for it.

The loop below is rather contrived. We would really just use *cat* command to see the contents. But it's a good practice because the output is predictable.

```
In [39]: %%bash
         root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

         while read f;do
             echo $f
         done < $root_dir/ext_list.txt
```

```
6 "conf"
6 "xsl"
7 "gif"
7 "H16"
7 "pdf"
7 "sa"
9 "h_shipped"
11 "c_shipped"
13 "tst"
14 "ppm"
15 "lds"
23 "pl"
26 "HEX"
28 "debug"
33 "tmpl"
34 "sh"
50 "boot"
79 "gitignore"
105 "xml"
111 "ihex"
115 "dts"
857 "txt"
1080 "S"
2818 "NONE"
11638 "h"
13154 "c"
```

Bash will delimit the output of the cat, by spaces or newlines

`(\n)`

. So in order to get the output as we'd like, we need each line to be delimited by

`\n`

, thus the IFS syntax.

Read is a better way to work with the contents of a file where it's assumed that you want data on a per-line basis. Most of the time we do.

```
In [41]: %%bash
         root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

         IFS=$'\n'
```

```
        for f in $(cat $root_dir/ext_list.txt);do
            echo $f
        done
```

```
      6 "conf"
      6 "xsl"
      7 "gif"
      7 "H16"
      7 "pdf"
      7 "sa"
      9 "h_shipped"
     11 "c_shipped"
     13 "tst"
     14 "ppm"
     15 "lds"
     23 "pl"
     26 "HEX"
     28 "debug"
     33 "tmpl"
     34 "sh"
     50 "boot"
     79 "gitignore"
    105 "xml"
    111 "ihex"
    115 "dts"
    857 "txt"
   1080 "S"
   2818 "NONE"
  11638 "h"
  13154 "c"
```

Let's do something slighly more interesting and introduce the *test* while were at it.

```
In [42]: %%bash
         root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

         while read f;do
                 count=$(echo $f | awk '{print $1}')
             if [ $count -gt 10 ];then
                 echo $f
             fi
         done < $root_dir/ext_list.txt
```

```
11 "c_shipped"
13 "tst"
14 "ppm"
15 "lds"
23 "pl"
26 "HEX"
28 "debug"
33 "tmpl"
34 "sh"
50 "boot"
79 "gitignore"
105 "xml"
```

```
111 "ihex"
115 "dts"
857 "txt"
1080 "S"
2818 "NONE"
11638 "h"
13154 "c"
```

10.9.2 Use case

Maybe you run a program in a loop, and save each output to a new file named with the dat and time

```
In [43]: %%bash
         root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

         for ind in {1..5};do
             fname="${root_dir}/$(date +%Y_%m_%d_%H:%M:%S).test"
             echo $fname
         done
```

```
command_line_pres_data/2015_09_30_19:07:04.test
command_line_pres_data/2015_09_30_19:07:04.test
command_line_pres_data/2015_09_30_19:07:04.test
command_line_pres_data/2015_09_30_19:07:04.test
command_line_pres_data/2015_09_30_19:07:04.test
```

```
In [44]: %%bash
         root_dir="/home/daniel/git/Python2.7/DataScience/command_line_pres_data"

         for ind in {1..5};do
             fname="${root_dir}/test_$ind.txt"
             echo $fname
         done
```

```
command_line_pres_data/test_1.txt
command_line_pres_data/test_2.txt
command_line_pres_data/test_3.txt
command_line_pres_data/test_4.txt
command_line_pres_data/test_5.txt
```

11 You made it pretty damned far.... I've run out of things to cram into this tutorial.