Command Line Cont.

We continue our discussion on command line today. The material we use are drawn from https://www.datacamp.com/courses/introduction-to-shell-for-data-science and https://www.howtogeek.com/107217/how-to-manage-processes-from-the-linux-terminal-10-commands-you-need-to-know/.

Review

We do a short review of a few things we learned yesterday.

Basics

- printing text: `echo 'text'`
- editing line : `^A` , `^E`

Manipulating files

- Redirecting and Appending: Redirect `> filename` , Appending `>>` , and Inspecting: `cat`
- Listing `ls` , moving `mv` , copying `cp` and deleting `rm` . What is `ls -a` ? Autocompletion `tab` .

Inspecting Files

- Installing and Downloading: `sudo apt install program_name` , `curl -OL url`
- `head` , `tail` , `wc` and pipelines `|` .
- Searching text `grep text textfile`
- Read a file `less` , Help `man` .

Directories

- Current path `.` , home directory `~` , and root directory `/` , One directory up `..` , Checking where you are `pwd` .

- Navigating directories `cd`, Creating Directory `mkdir`
- Renaming and moving `mv`, copying `cp` and deleting `rm -rf` and `rmdir`.

Exercise

1. Switch to your home directory and then create directories called `tmp` and `scratch`
2. Go into the `tmp` directory and download the file `https://people.orie.cornell.edu/bdg79/Seeing_Off_a_Friend.txt` with name `Seeing_Off_a_Friend.txt` to your current directory. Use `head` and `tail` and `|` to extract the third and fourth line of `Seeing_Off_a_Friend.txt` and output (using `>`) these two line to a file named `Seeing_Off`.
3. Now switch to the home directory. List the files in `tmp` without switching to it.
4. Go in to the `scratch` directory. Copy the file `Seeing_Off` to `scratch` (You might want to use `..` and `.`).
5. Go to the home directory, delete the two directories `tmp` and `scratch`.

Data Manipulation

It would be helpful if we could do some preliminary analysis of the data before using more powerful tools in python.

Create a directory called `roster`. Go into `roster` and Download the file `https://people.orie.cornell.edu/bdg79/data.txt`.
The file `data` is of the format

```
name, phone Number, email, age.
```

We shall work with this file and this directory in this section.

`cut`

default is tab

The `cut` function allows us to select columns of a text file. For example, suppose we want to select the first and second columns of data, we can do

```
cut -f 1-2 -d , data.txt | head -5
```

Here `-f 1-2` specifies the columns we want and `-d ,` means use the delimiter `,`. The default is `\t` which is `tab` and one common choice of delimiter is space `' '`.

`sort`

It is a good habit to put things in order and the `sort` command allows us to do so. Let us sort the `data` by name in ascending alphabetical order, (a to z):

> ```
> sort data.txt
> ```

| head -5

If you want to sort in reverse, put the option `-r` there. The option (or flag) `-n` means to sort numerically, while `-b` tells `sort` to ignore leading blanks and `-f` tells `sort` to **f**old case (i.e., be case-insensitive).

sort -r data.txt

`uniq`

Another command that is often used with `sort` is `uniq`, whose job is to remove duplicated lines. More specifically, it removes *adjacent* duplicated lines. If a file contains:

```
2017-07-03
2017-07-03
2017-08-03
2017-08-03
```

then `uniq` will produce:

```
2017-07-03
2017-08-03
```

but if it contains:

```
2017-07-03
2017-08-03
2017-07-03
2017-08-03
```

then `uniq` will print all four lines. The reason is that `uniq` is built to work with very large files. In order to remove non-adjacent lines from a file, it would have to keep the whole file in memory (or at least, all the unique lines seen so far). By only removing adjacent duplicates, it only has to keep the most recent unique line in memory.

`uniq` has an option `-c` which display unique lines with a count of how often each occurs. Suppose a text file `date.txt` contains the following dates.

```
2017-07-03
2017-08-03
2017-07-03
2017-08-03
```

Then the command line

> `uniq -c date.txt`

outputs the following.

```
1 2017-07-03
1 2017-08-03
1 2017-07-03
1 2017-08-03
```

To make `uniq` work as we wanted, it is helpful to combine it with `sort`, so

> `sort date.txt | uniq -c`

same date is adjencent

gives

```
2 2017-07-03
2 2017-08-03
```

which is what we want.

Apart from `sort`, `cut` and `uniq`. `wc` is also often used in data manipulation.

Exercise *cut -f 1 -d, data.txt | cut -f 1 -d ' ' | sort | uniq | wc*

Figure out the following questions using command line:

1. How many people are listed in `data`? *grep '(' data.txt | wc*
2. How many different first names are there? How about second name?
3. How many people write their phone number in the form (###)###-###?
4. How many different domain names of email are there? Domain name means the part after the `@` sign. *cut -f 3 -d, data.txt | cut -f 2 -d "@" | sort | uniq -c*
5. What is the range of the age, i.e., what is the largest/smallest number in the last column? *| sort -n | head -1 or tail -1*
6. How many different ages are there? Which age has the highest frequency? Which age has the lowest frequency? *sort| uniqu -c | sort | tail -1*

7. How many people has first name started with `Er` ? (`grep '^Er'` would be helpful)
   How many people has last name end with `son` ? (`grep 'son$'` would be helpful)
8. Which domain name appears most often? | *grep '^ Er' | wc*
9. Which first name appears most often and which last name has the least appearance?
10. How many phone number has `110` in it?
11. Are your instructors' name in the roster, i.e., can you find 'Lijun Ding' and 'Ben Grimmer'? *grep 'Lijun Ding' data.txt*

Once you finish the above exercise, create a directory `DomainName` and go into it. Create n files each contains the lines with the same domain name in the email address. Here n (which should be 4) is the number of different domain names. Name each file with the domain name but with `.` replaced by `_`. For example, `orie_cornell_edu.txt` should contain all the lines in `data.txt` whose email address has domain name `orie.cornell.edu`. *mkdir DomainName*
cd DomainName/

Wildcards

Actually, many of our previous command can applied to many files together and that is why command line is useful! (Allows you to manipulate a lot of files together)
For example if you type

paste the columns together

> ```
> cut -d , -f 1 orie_cornell_edu.txt gmail_txt yahoo_txt hotmail_com
> ```

Then you should get all the names of `data.txt`. However, it is a bit troublesome if we need to type every file name. Is there a way to avoid that?

To make your life better, the command line allows you to use **wildcards** to specify a list of files with a single expression. The most common wildcard is `*`, which means "match zero or more characters". Using it, we can shorten the `cut` command above to be:

```
cut -d , -f 1 *
```
anything in your current directory 很上面的效果是一样的

What other wildcards can I use?

The shell has other wildcards as well, though they are less commonly used:

- `?` matches a single character, so `201?.txt` will match `2017.txt` or `2018.txt`, but not `2017-01.txt`.

only one character

- `[...]` matches any one of the characters inside the square brackets, so `201[78].txt` matches `2017.txt` or `2018.txt`, but not `2016.txt`.
- `{...}` matches any of the comma-separated patterns inside the curly brackets, so `{*.txt, *.csv}` matches any file whose name ends with `.txt` or `.csv`, but not files whose names end with `.pdf`.

Which expression in the following would match `singh.pdf` and `johel.txt` but *not* `sandhu.pdf` or `sandhu.txt`?

1. `[sj]*.{.pdf, .txt}`
2. `{s*.pdf, j*.txt}`
3. `[singh,johel]{*.pdf, *.txt}`
4. `{singh.pdf, j*.txt}`

Exercise

Switch to the `DomainName` directory. Make sure you have all four files. You job is to build a pipeline to find out which files has the smallest number of lines. Your output should be of the form:

> numberOfLines filename

You can utilize the following procedures.

1. Use `wc` with appropriate parameters to list the number of lines *but not word and bytes* in all of four files. (Use a wildcard for the filenames instead of typing them all in by hand. To match the requirement that *showing lines but not word and bytes*, try `man wc` and see which option you can use)
2. Add another command to the previous one using a pipe to remove the line containing the word "total". (Use `man grep` and search `invert`)
3. Add two more stages to the pipeline that use `sort` and `head` to find the file containing the fewest lines. (You should put options to both commands)

Batch processing and Create your own commands

Most commands will process many files at once as we have just seen. You now will learn how to make your own pipelines do that (so that you have more precise control when you need). Along the way, you will see how our command line program (shell) uses variables to store information.

Variables

Environment variables

Like other programs, the shell stores information in variables. Some of these, called **environment variables**, are available all the time. Environment variables' names are conventionally written in upper case, and a few of the more commonly-used ones are shown below.

Variable	Purpose	Value
`HOME`	User's home directory	Same as `echo ~`
`PWD`	Present working directory	Same as `pwd` command
`SHELL`	Which shell program is being used	`/bin/bash`
`USER`	User's ID	Same as `echo $USER`

To get a complete list (which is quite long), you can type `set` in the shell.

To print the value of a variable `X`:

> `echo $X

Local variables

Apart from environment variables which is set by the system, you can create your variables which *only lasts in your current shell*. Suppose you want the name of the variable being `cornell` and the value being `orie_cornell_edu`, you can achieve this via

> `cornell=orie_cornell_edu`

Note that there is **no spaces** before or after the `=` sign! You can achieve the value of `cornell` via

> `$cornell`

Question Will `head orie_cornell_edu` and `head $cornell` produce the same result?

For Loop!

The loop structure will allow us to manipulate a lot files together. Let us start with something simple, try to run

```
for filetype in gif jpg png; do echo $filetype; done
```

Notice these things about the loop:

1. The structure is `for` …variable… `in` …list… `;` `do` …body… `;` `done`
2. The list of things the loop is to process (in our case, the words `gif` , `jpg` , and `png`).
3. The variable that keeps track of which thing the loop is currently processing (in our case, `filetype`).
4. The body of the loop that does the processing (in our case, `echo $filetype`). What if you type

```
for filetype in gif jpg png; do echo filetype; done ?
```

Now suppose we want to echo the name of all the files in our current directory via for loop. You can type

filetype

```
for filename in *; do echo $filename; done
```

To list all files in directory up and use variables concept: you can type

```
updir=../*
for filename in $updir; do echo $filename; done
```

You can do many things in one loop via `;` after `do` . For example,

```
for filename in *; do echo $filename ; head -1 $filename; done ,
```

which prints the file name and print the first line of the file.

Creating your own command!

It would be better if you can write down specific command and run them later without typing the whole thing once again. As you can see from the loop, it could be quite complicated if you want to multiple things
and also creating a long pipelines. Making everything in a line is error prone. Creating a command line file can make your life a bit better.

Create a `.sh` file

To start, type

> `nano iterate.sh`

Then we type

```
for filename in *;
  do echo $filename ;
  head -1 $filename;
done
```

in the `iterate.sh` file. In order to run the file, type

> `bash iterate.sh`.

You can record the output via `>`. You can actually avoid the `;` by

```
for filename in *
do
  echo $filename
  head -1 $filename
done
```

Input arguments

Our script would be more useful if it can have input arguments. To do that, we use the expression `$@`. Recall we can count distinct elements via

> `sort filename | uniq -c`

We can create a `unique-lines.sh` to record this command:

> `sort $@ | uniq -c`

then when you run:

> `bash unique-lines.sh date1.txt`

the shell replaces `$@` with `date1.txt` and processes one file. If you run this:

> `bash unique-lines.sh date1.txt date2.txt`

it processes two data files, and so on.

You can even have more precise arguments by `$1` and `$2`. For example, you can create a script called `column.sh` that selects a single column from a CSV file when the user provides the filename as the first parameter and the column as the second:

> `cut -d , -f $2 $1`

and then run it using:

> `bash column.sh gmail_com 1`

Notice how the script uses the two parameters in reverse order.

Many commands in one `.sh` file

You can have multiple commands in one `.sh` file. For example, you can create a `combined.sh` with contents

```
cut -f 4 -d , $@ | sort -n | uniq -c | head -1
for filename in $@
do
  echo $filename
  head -1 $filename
done
```

Exercise

Create a `.sh` file so that it contains a loop and `$@` to do the task of showing the age of the youngest and oldest people for `orie_cornell_edu.txt`, `yahoo_com.txt`, `gmail_com.txt`, and `hotmail.txt`. You can suppose they are all in the same directory and the directory contains only these four files (so you can use the wildcard `*`) and you run the `.sh` in this directory. Your output should be something like

```
file_name1
age of youngest person of file_name1
oldest of youngest person of file_name1
file_name2
age of youngest person of file_name2
oldest of youngest person of file_name2
....
```

Processes Management

`top` , `ps` , and `kill`

The `top` command is the traditional way to view your system's resource usage and see the processes that are taking up the most system resources. Top displays a list of processes, with the ones using the most CPU at the top. Use `^C` to get rid of `top` .

The `ps` command lists running processes. The following command lists all processes running on your system:

```
ps -A    | less
```

One can use `grep` to search for certain applications.
The most important thing is the `PID` , an identification number for the process. You can kill the program via

```
kill PID
```

If it fails, try

```
kill -KILL PID
```

Run program in the background

Sometimes you want to use command line to run certain programs but normally, if you run the program, you can not use command line any more. The way to get around this is to type `@` in the end of the command.

```
command ~~@~~. &
```

You can check jobs in the background by `jobs -l` and kill them in the same way as before.