Intro to Shell for Data Science - Greg Wilson

Note as of Dec 2019
Note Taker: Paris Zhang

Ch 1 - Manipulating files and directories

1. The command line begins with `/` is an **absolute path**, otherwise it is a **relative path**.
2. **Move around directory**: `..` - one directory above, `.` - the current directory, `~` - the home directory.
3. **Copy files** (or overwrite): `cp original.txt duplicate.txt`.
4. **Copy multiple files** to a directory: `cp seasonal/autumn.csv seasonal/winter.csv backup`.
5. **Move a file**: `mv autumn.csv winter.csv ..` (move multiple files to one directory above).
6. **Rename files** (or overwrite): `mv course.txt old-course.txt`.
7. **Delete files**: `rm thesis.txt backup/thesis-2017-08.txt`.
8. If in the home directory and run `mv seasonal by-season`, then `mv` changes the name of the directory.
9. **Delete directory**: use `rmdir` or `rm -r`.
10. **Create directory**: use `mkdir`.

Ch 2 - Manipulating data

1. **Concatenate**: `cat agarwal.txt`, which prints all file contents and may be viewed by scrolling
2. **View contents by page**: `less` a file and then one page is displayed at a time; press spacebar to page down or type `q` to quit. If `less` several files, type `:n` to move to the next file, `:p` to go back to the previous one, or `:q` to quit.
3. **Display** the `head` of a file (10 lines), or adding a **command-line flag** (or just "flag" for short) to view the top 3 lines: `head -n 3 seasonal/summer.csv`. Command flags don't have to be a `-` followed by a single letter, but it's a widely-used convention. Note: it's considered good style to put all flags before any filenames.
4. Use `tail` with the flag `-n +7` to display all but the first six lines of `seasonal/spring.csv`.
5. **List everything** below a directory: `ls -R` (this shows every file and directory in the current level, then everything in each sub-directory, and so on), or `ls -RF` to print a `/` after the name of every directory and a `*` after the name of every runnable program.

6. Get help for a command by using `man` (short for "manual")
   `man head`

`man` automatically invokes `less`, press spacebar to page through the information and `:q` to quit. The one-line description under `NAME` tells you briefly what the command does, and the summary under `SYNOPSIS` lists all the flags it understands. Anything that is optional is shown in square brackets `[...]`, either/or alternatives are separated by `|`, and things that can be repeated are shown by `...`, so `head`'s manual page is telling you that you can *either* give a line count with `-n` or a byte count with `-c`, and that you can give it any number of filenames.	see output below

```
HEAD(1)                 BSD General Commands Manual                HEAD(1)

NAME
     head -- display first lines of a file

SYNOPSIS
     head [-n count | -c bytes] [file ...]

DESCRIPTION
     This filter displays the first count lines or bytes of each of
     the specified files, or of the standard input if no files are
     specified.  If count is omitted it defaults to 10.

     If more than a single file is specified, each file is preceded by
     a header consisting of the string ``==> XXX <=='' where ``XXX''
     is the name of the file.

SEE ALSO
     tail(1)
```

7. **Select columns**: `cut -f 2-5,8 -d , values.csv` to select columns 2 through 5 and columns 8, using comma as the separator. `cut` uses `-f` (meaning "fields") to specify columns and `-d` (meaning "delimiter") to specify the separator. You need to specify the latter because some files may use spaces, tabs, or colons to separate columns.
   `cut -d, -f1 seasonal/spring.csv` = `cut -d , -f 1 seasonal/spring.csv`.
8. **Repeat commands**: `history` will print a list of commands you have run recently. Each one is preceded by a serial number: type `!55` to re-run the 55th command in the history. Re-run a command by typing an exclamation mark followed by the command's name, such as `!head` or `!cut`, which will re-run the most recent use of that command.
9. **Select lines containing specific values**: `grep bicuspid seasonal/winter.csv` prints lines from `winter.csv` that contain `"bicuspid"`.
   `grep` most common flags:
   - `-c` : print a count of matching lines rather than the lines themselves
   - `-h` : do *not* print the names of files when searching multiple files
   - `-i` : ignore case (e.g., treat "Regression" and "regression" as matches)
   - `-l` : print the names of files that contain matches, not the matches
   - `-n` : print line numbers for matching lines
   - `-v` : invert the match, i.e., only show lines that *don't* match

---

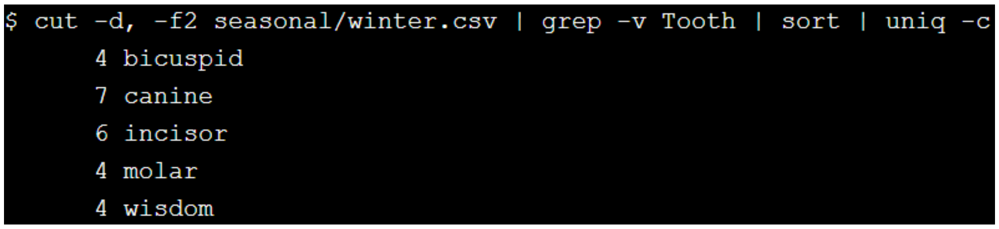
Ch 3 - Combining tools

1. **Store an output to a file**: `head -n 5 seasonal/summer.csv > top.csv`, then `head`'s output is put in a new file called `top.csv`. The file can be viewed by `cat top.csv`.
2. **Combine commands**: use a pipe symbol `|`, e.g., `head -n 5 seasonal/summer.csv | tail -n 3`.
3. **Chain commands**:
   `cut -d , -f 1 seasonal/spring.csv | grep -v Date | head -n 10` will
   a. Select the first column from the spring data;
   b. Remove the header line containing the word "Date"; and
   c. Select the first 10 lines of actual data.
4. **Count** the number of **c**haracters, **w**ords, and **l**ines using `wc` with `-c`, `-w`, or `-l` respectively. E.g., `grep 2017-07 seasonal/spring.csv | wc -l` prints the count of lines that contains the date 2017-07 in spring.csv.

5. **Specify a list of files** using wildcards like `*`, which means "match zero or more characters".
6. **Common wildcards**:

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

7. **Sort text** using `sort` (ascending order by default):
   `-n`: sort numerically
   `-r`: reverse the order
   `-b`: ignore leading blanks
   `-f`: fold case (i.e., case-insensitive)
8. **Remove duplicate lines**: write a pipeline to
   a. get the second column from `seasonal/winter.csv`,
   b. remove the word "Tooth" from the output so that only tooth names are displayed,
   c. sort the output so that all occurrences of a particular tooth name are adjacent; and
   d. display each tooth name once along with a count of how often it occurs.

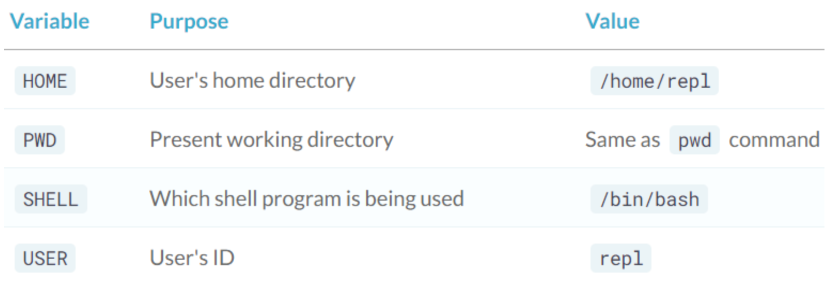


```
$ cut -d, -f2 seasonal/winter.csv | grep -v Tooth | sort | uniq -c
      4 bicuspid
      7 canine
      6 incisor
      4 molar
      4 wisdom
```

9. **Stop a running program**: type `Ctrl + c`, looks like `^C` in the console.

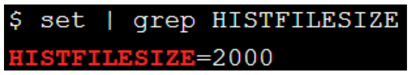
Ch 4 - Batch processing

1. Shell stores information in variables. Some of these, called **environment variables**, are available all the time. Get a full list using `set`. Some commonly-used ones:



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

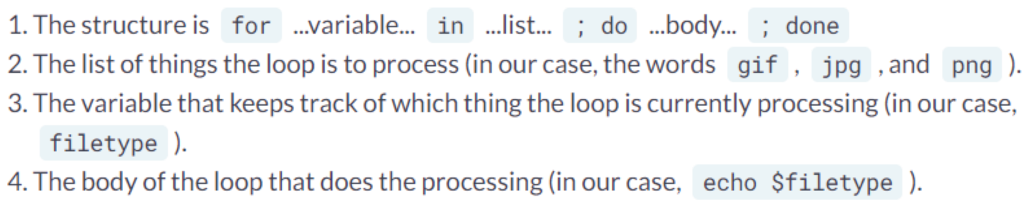
2. Use `set` and `grep` with a pipe to display the value of `HISTFILESIZE`, which determines how many old commands are stored in your command history.



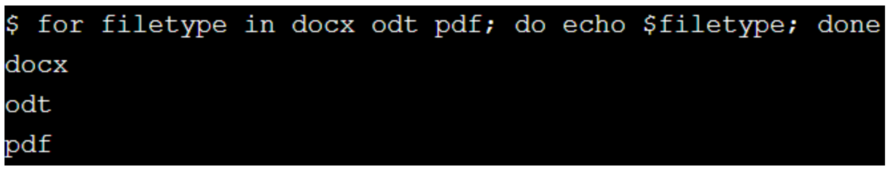
```
$ set | grep HISTFILESIZE
HISTFILESIZE=2000
```

3. **Print a variable's value** using `echo $USER`, which prints `repl`. This is true everywhere: to get the value of a variable called `X`, you must write `$X`. (This is so that the shell can tell whether you mean "a file named X" or "the value of a variable named X").
4. **Create a shell variable**: `training=seasonal/summer.csv` (no space).

5. **Repeat a command** with **loops**:

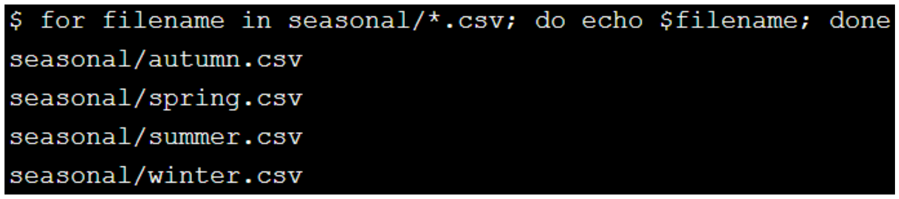


1. The structure is `for` ...variable... `in` ...list... `; do` ...body... `; done`
2. The list of things the loop is to process (in our case, the words `gif`, `jpg`, and `png`).
3. The variable that keeps track of which thing the loop is currently processing (in our case, `filetype`).
4. The body of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



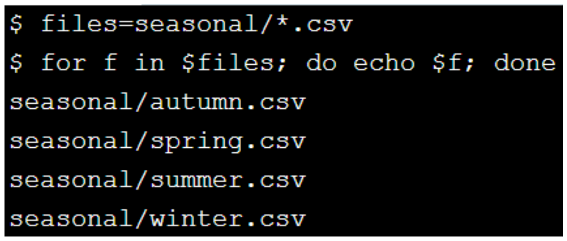
```
$ for filetype in docx odt pdf; do echo $filetype; done
docx
odt
pdf
```

6. **Repeat a command for multiple files**:



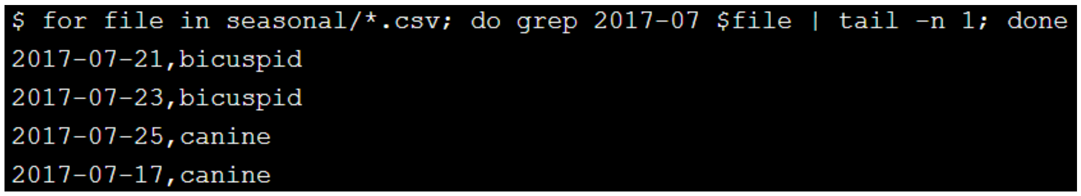
```
$ for filename in seasonal/*.csv; do echo $filename; done
seasonal/autumn.csv
seasonal/spring.csv
seasonal/summer.csv
seasonal/winter.csv
```

7. **Record names of multiple files**:



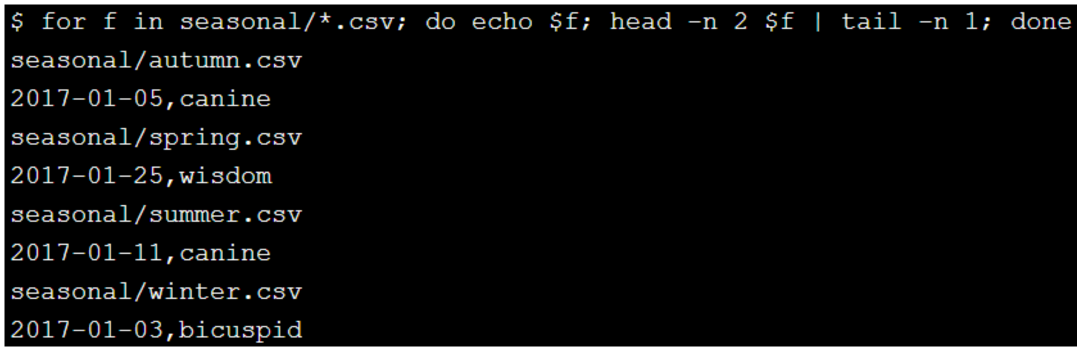
```
$ files=seasonal/*.csv
$ for f in $files; do echo $f; done
seasonal/autumn.csv
seasonal/spring.csv
seasonal/summer.csv
seasonal/winter.csv
```

8. **Loop pipeline**:



```
$ for file in seasonal/*.csv; do grep 2017-07 $file | tail -n 1; done
2017-07-21,bicuspid
2017-07-23,bicuspid
2017-07-25,canine
2017-07-17,canine
```

9. **Many commands in a single loop**:

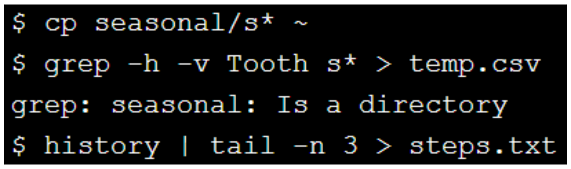


```
$ for f in seasonal/*.csv; do echo $f; head -n 2 $f | tail -n 1; done
seasonal/autumn.csv
2017-01-05,canine
seasonal/spring.csv
2017-01-25,wisdom
seasonal/summer.csv
2017-01-11,canine
seasonal/winter.csv
2017-01-03,bicuspid
```

Ch 5 - Creating new tools

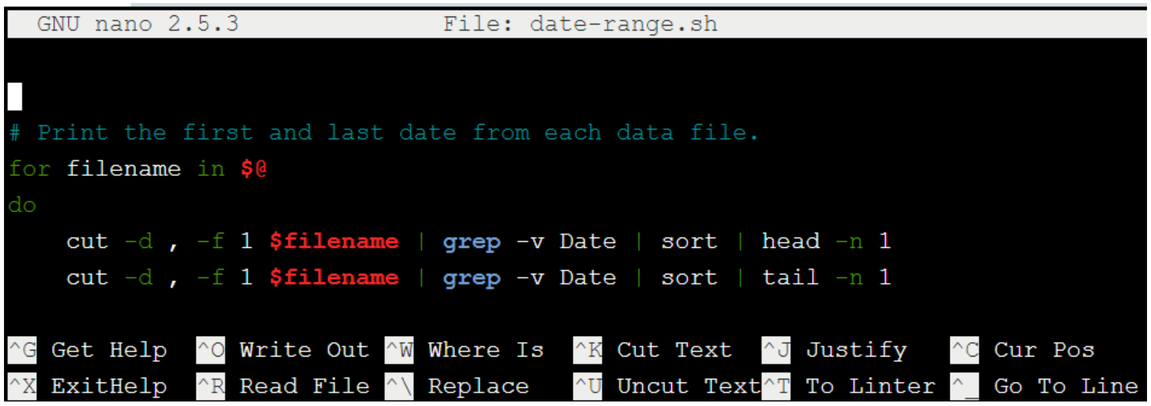
1. **Edit a file**: `nano filename`, which will open filename for editing (or create it if it doesn't already exist).
    a. Ctrl + K: delete a line.

   b. Ctrl + U: un-delete a line.
   c. Ctrl + O: save the file ('O' stands for 'output').
   d. Ctrl + X: exit the editor.
2. **Record recent commands**:
   a. Run `history`.
   b. Pipe its output to `tail -n 10` (or however many recent steps you want to save).
   c. Redirect that to a file called something like `figure-5.history`.



```
$ cp seasonal/s* ~
$ grep -h -v Tooth s* > temp.csv
grep: seasonal: Is a directory
$ history | tail -n 3 > steps.txt
```

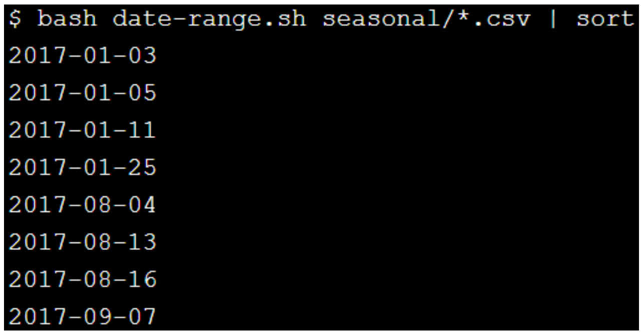
3. **Save commands to re-run** later: use `nano headers.sh` to create a shell script file that contains the command `head -n 1 seasonal/*.csv`, then run it by `bash headers.sh`.
4. **Pass filenames to scripts**: if `unique-lines.sh` contains `sort $@ | uniq`, when you run: `bash unique-lines.sh seasonal/summer.csv`, the shell replaces `$@` with `seasonal/summer.csv` and processes one file. If you run this: `bash unique-lines.sh seasonal/summer.csv seasonal/autumn.csv`, it processes two data files, and so on.
5. **Process a single argument**: As well as `$@`, the shell lets you use `$1`, `$2`, and so on to refer to specific command-line parameters. The script `get-field.sh` contains `head -n $2 $1 | tail -n 1 | cut -d , -f $3`, then `bash get-field.sh seasonal/summer.csv 4 2` should select the second field from line 4 of `seasonal/summer.csv`.
6. **Loops in shell script**:
   `nano date-range.sh`



```
  GNU nano 2.5.3                 File: date-range.sh


# Print the first and last date from each data file.
for filename in $@
do
    cut -d , -f 1 $filename | grep -v Date | sort | head -n 1
    cut -d , -f 1 $filename | grep -v Date | sort | tail -n 1

^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X ExitHelp   ^R Read File  ^\ Replace    ^U Uncut Text^T To Linter   ^_ Go To Line
```



```
$ bash date-range.sh seasonal/*.csv | sort
2017-01-03
2017-01-05
2017-01-11
2017-01-25
2017-08-04
2017-08-13
2017-08-16
2017-09-07
```