Linux CheatSheet

On linux, unlike the VAX, file names and directory names are case sensitive. You can tab complete instead of typing in the whole filename/directory name. Press tab twice to list possible completions.

Working with files and directories from the Linux commandline:

Command	Description
pwd	Show current directory
cd ~	Change to your home directory (same as your W: drive on WIndows)
cd *dirname*	Change directory to a subfolder named *dirname*
cd ..	Go up a directory
ls	List files and folders in the current directory (with color highlighting)
dir	List files and folders in the current directory without color highlighting
mkdir *dirname*	Make a new subfolder named *dirname*
rmdir *dirname*	Deletes the subfolder name *dirname* (forlder must be empty)
rm *filename*	Deletes the file named *filename*
rm -rf *dirname*	Recursively deletes the non-empty folder name *dirname* and all contained files and folders
mv *oldname newname*	Rename a file or directory from *oldname* to *newname*
mv *src destination*	Move a file or directory from *src* to *destination*
cp *src destination*	Copies a **file** from *src* to *destination*
cp -av *src* destination	Recursively copies a **folder** from *src* to *destination*
xdg-open .	Open a gui file browser in the current directory
xdg-open *file*	Open a file

Command Line Tricks for Data Scientists

This is reproduced from: https://medium.com/@kadek/command-line-tricks-for-data-scientists-c98e0abe5da .

ICONV

File encodings can be tricky. For the most part files these days are all UTF-8 encoded. To understand some of the magic behind UTF-8, check out this excellent video. Nonetheless, there are times where we receive a file that isn’t in this format. This can lead to some wonky attempts at swapping the encoding schema. Here, `iconv` is a life saver. Iconv is a simple program that will take text in one encoding and output the text in another.

```
# Converting -f (from) latin1 (ISO-8859-1)
# -t (to) standard UTF_8
iconv -f ISO-8859-1 -t UTF-8 < input.txt > output.txt
```

Useful options:

```
# List all known encodings
iconv -l

# Silently discard characters that cannot be converted
iconv -c
```

HEAD

Often when dealing with new data the first thing we want to do is get a sense of what exists. Head, without any flags, will print out the first 10 lines of a file. The true power of head lies in testing out cleaning operations. For instance, if we wanted to change the delimiter of a file from commas to pipes.

```
# Prints out first 10 lines
head filename.csv

# Prints first 3 lines
head -n 3 filename.csv
```

Useful options:

```
# Print a specific number of lines
head -n

# Print a specific number of bytes
head -c
```

TR

Tr is analogous to translate. This powerful utility is a workhorse for basic file cleaning. An ideal use case is for swapping out the delimiters within a file.

```
# Converting a tab delimited file into commas
cat tab_delimited.txt | tr "\\t" "," comma_delimited.csv
```

Another feature of tr is all the built in [:class:] variables at your disposal. These include:

```
[:alnum:] all letters and digits
[:alpha:] all letters
[:blank:] all horizontal whitespace
[:cntrl:] all control characters
[:digit:] all digits
[:graph:] all printable characters, not including space
[:lower:] all lower case letters
[:print:] all printable characters, including space
[:punct:] all punctuation characters
[:space:] all horizontal or vertical whitespace
[:upper:] all upper case letters
[:xdigit:] all hexadecimal digits
```

You can chain a variety of these together to compose powerful programs. The following is a basic word count program you could use to check your READMEs for overuse.

```
cat README.md | tr "[:punct:][:space:]" "\n" | tr "[:upper:]" "[:lower:]" | grep . | sort | uniq -c | sort -nr
```

Another example using basic regex:

```
# Converting all upper case letters to lower case
cat filename.csv | tr '[A-Z]' '[a-z]'
```

Useful options:

```
# Delete characters
tr -d

# Squeeze characters
tr -s

# Backspace
\b

# Form feed
\f

# Vertical tab
\v

# Characters with octal value NNN
\NNN
```

WC

Word count. Its value is primarily derived from the -l flag, which will give you the line count.

```
# Will return number of lines in CSV
wc -l gigantic_comma.csv
```

This tool comes in handy to confirm the output of various commands. So, if we were to convert the delimiters within a file and then run wc -l we would expect the total lines to be the same. If not, then we know something went wrong.

Useful options:

```
# Print the byte counts
wc -c

# Print the character counts
wc -m

# Print length of longest line
wc -L

# Print word counts
wc -w
```

SPLIT

File sizes can range dramatically. And depending on the job, it could be beneficial to split up the file—thus split. The basic syntax for split is:

```
# We will split our CSV into new_filename every 500 lines
split -l 500 filename.csv new_filename_


# filename.csv

# new_filename_aaa
# new_filename_aab
# new_filename_aac
```

Two quirks are the naming convention and lack of file extensions. The suffix convention can be numeric via the -d flag. To add file extensions, you'll need to run the following *find* command. It will change the names of ALL files within the current directory by appending .csv, so be careful.

```
find . -type f -exec mv '{}' '{}'.csv \;


# filename.csv.csv
# new_filename_aaa.csv
# new_filename_aab.csv
# new_filename_aac.csv
```

Useful options:

```
# Split by certain byte size
split -b

# Generate suffixes of length N
split -a

# Split using hex suffixes
split -x
```

SORT & UNIQ

The preceding commands are obvious: they do what they say they do. These two provide the most punch in tandem (i.e. unique word counts). This is due to *uniq*, which only operates on duplicate adjacent lines. Thus, the reason to sort before piping the output through. One interesting note is that sort -u will achieve the same results as the typical *sort file.txt* | *uniq* pattern.

Sort does have a sneakily useful ability for data scientists: the ability to sort an entire CSV based on a particular column.

```
# Sorting a CSV file by the second column alphabetically
sort -t, -k2 filename.csv


# Numerically
sort -t, -k2n filename.csv


# Reverse order
sort -t, -k2nr filename.csv
```

The -t option here is to specify the comma as our delimiter. More often than not spaces or tabs are assumed. Furthermore, the -k flag is for specifying our key.

Useful options:

```
# Ignore case
sort -f

# Reverse sort order
sort -r

# scramble order
sort -R

# Count number of occurrences
uniq -c

# Only print duplicate lines
uniq -d
```

CUT

Cut is for removing columns. To illustrate, if we only wanted the first and third columns:

```
cut -d, -f 1,3 filename.csv
```

To select every column other than the first:

```
cut -d, -f 2- filename.csv
```

In combination with other commands, *cut* serves as a filter:

```
# Print first 10 lines of column 1 and 3, where "some_string_value" is present
head filename.csv | grep "some_string_value" | cut -d, -f 1,3
```

Finding out the number of unique values within the second column:

```
cat filename.csv | cut -d, -f 2 | sort | uniq | wc -l


# Count occurences of unique values, limiting to first 10 results
cat filename.csv | cut -d, -f 2 | sort | uniq -c | head
```

PASTE

Paste is a niche command with an interesting function. If you have two files that you need merged, and they are already sorted, paste has you covered.

```
# names.txt
adam
john
zach

# jobs.txt
lawyer
youtuber
developer

# Join the two into a CSV
paste -d ',' names.txt jobs.txt > person_data.txt

# Output
adam,lawyer
john,youtuber
zach,developer
```

JOIN

Join is a simplistic, quasi-tangential, SQL. The largest differences being that joinwill return all columns and matches can only be on one field. By default, join will try and use the first column as the match key. For a different result, the following syntax is necessary:

```
# Join the first file (-1) by the second column
# and the second file (-2) by the first
join -t, -1 2 -2 1 first_file.txt second_file.txt
```

The standard join is an inner join. However, an outer join is also viable through the *-a* flag. Another noteworthy quirk is the *-e* flag, which can be used to substitute a value if a missing field is found.

```
# Outer join, replace blanks with NULL in columns 1 and 2
# -o which fields to substitute - 0 is key, 1.1 is first column, etc...
join -t, -1 2 -a 1 -a2 -e ' NULL' -o '0,1.1,2.2' first_file.txt second_file.txt
```

Useful options:

```
# Print unpairable lines
join -a

# Replace missing input fields
join -e

# Equivalent to -1 FIELD -2 FIELD
join -j
```

GREP

Global search for a regular expression and print, or grep; likely, the most well known command, and with good reason. Grep has a lot of power, especially for finding your way around large codebases. Within the realm of data science, it acts as a refining mechanism for other commands. Although its standard usage is valuable as well.

```
# Recursively search and list all files in directory containing 'word'
grep -lr 'word' .


# List number of files containing word
grep -lr 'word' . | wc -l
```

Count total number of lines containing word/pattern

```
grep -c 'some_value' filename.csv


# Same thing, but in all files in current directory by file name
grep -c 'some_value' *
```

Grep for multiple values using the *or* operator – \|

```
grep "first_value\|second_value" filename.csv
```

Useful options:

```
# Make grep output colorful
alias grep="grep --color=auto"

# Use extended regular expressions
grep -E

# Only match whole words
grep -w

# Print name of files with match
grep -l

# Inverted matching
grep -v
```

SED

At its core sed is a stream editor that operates on a line-by-line basis. It excels at substitutions, but can also be leveraged for all out refactoring.

The most basic *sed* command consists of *s/old/new/g* . This translates to search for old value, replace all occurences in-line with new. Without the */g* our command would terminate after the first occurrence on the line.

To get a quick taste of the power lets dive into an example. In this scenario you've been given the following file:

```
balance,name
$1,000,john
$2,000,jack
```

The first thing we may want to do is remove the dollar signs. The *-i* flag indicates in-place. The two single quotes are to indicate a zero-length file extension, thus overwriting our initial file. Ideally, you would test each of these individually and then output to a new file.

```
sed -i '' 's/\$//g' data.txt


# balance,name
# 1,000,john
# 2,000,jack
```

Next up, the commas in our balance column values:

```
sed -i '' 's/\([0-9]\),\([0-9]\)/\1\2/g' data.txt

# balance,name
# 1000,john
# 2000,jack
```

Lastly, Jack up and decided to quite one day. So, removing him:

```
sed -i '' '/jack/d' data.txt

# balance,name
# 1000,john
```

AWK

The best for last. Awk is much more than a simple command: it is a full-blown language. Of everything covered in this article, awk is by far the coolest.

Common use cases for *awk* include:

- Text processing
- Formatted text reports
- Performing arithmetic operations
- Performing string operations

Awk can parallel *grep* in its most nascent form:

```
awk '/word/' filename.csv
```

Or with a little more magic the combination of *grep* and *cut*. Here, *awk* prints the third and fourth column, tab separated, for all lines with our word. *-F*, merely changes our delimiter to a comma:

```
awk -F, '/word/ { print $3 "\t" $4 }' filename.csv
```

Awk comes with a lot of nifty variables built-in. For instance, *NF* - number of fields - and *NR* - number of records. To get the fifty-third record in a file:

```
awk -F, 'NR == 53' filename.csv
```

An added wrinkle is the ability to filter based off of one or more values. The first example, below, will print the line number and columns for records where the first column equals *string*.

```
awk -F, ' $1 == "string" { print NR, $0 } ' filename.csv


# Filter based off of numerical value in second column
awk -F, ' $2 == 1000 { print NR, $0 } ' filename.csv
```

Multiple numercial expressions:

```
# Print line number and columns where column three greater
# than 2005 and column five less than one thousand

awk -F, ' $3 >= 2005 && $5 <= 1000 { print NR, $0 } ' filename.csv
```

Sum the third column:

```
awk -F, '{ x+=$3 } END { print x }' filename.csv
```

The sum of the third column, for values where the first column equals “something”.

```
awk -F, '$1 == "something" { x+=$3 } END { print x }' filename.csv
```

Get the dimensions of a file:

```
awk -F, 'END { print NF, NR }' filename.csv


# Prettier version
awk -F, 'BEGIN { print "COLUMNS", "ROWS" }; END { print NF, NR }' filename.csv
```

Print lines appearing twice:

```
awk -F, '++seen[$0] == 2' filename.csv
```

Remove duplicate lines:

```
# Consecutive lines
awk 'a !~ $0; {a=$0}']


# Nonconsecutive lines
awk '! a[$0]++' filename.csv


# More efficient
awk '!($0 in a) {a[$0];print}
```

Substitute multiple values using built-in function *gsub()* :

```
awk '{gsub(/scarlet|ruby|puce/, "red"); print}'
```

This awk command will combine multiple CSV files, ignoring the header and then append it at the end.

```
awk 'FNR==1 && NR!=1{next;}{print}' *.csv > final_file.csv
```

Need to downsize a massive file? Welp, *awk* can handle that with help from *sed*. Specifically, this command breaks one big file into multiple smaller ones based on a line count. This one-liner will also add an extension.

```
sed '1d;$d' filename.csv | awk 'NR%NUMBER_OF_LINES==1{x="filename-"++i".csv";}{print > x}'


# Example: splitting big_data.csv into data_(n).csv every 100,000 lines
sed '1d;$d' big_data.csv | awk 'NR%100000==1{x="data_"++i".csv";}{print > x}'
```