grep, awk and sed

– three VERY useful command-line utilities

Matt Probert, Uni of York

grep = global regular expression print

In the simplest terms, grep (**g**lobal **r**egular **e**xpression **p**rint) will search input files for a search string, and print the lines that match it. Beginning at the first line in the file, grep copies a line into a buffer, compares it against the search string, and if the comparison passes, prints the line to the screen. Grep will repeat this process until the file runs out of lines. Notice that nowhere in this process does grep store lines, change lines, or search only a part of a line.

Example data file

Please cut & paste the following data and save to a file called 'a_file':

```
boot
book
booze
machine
boots
bungie
bark
aardvark
broken$tuff
robots
```

A Simple Example

The simplest possible example of grep is simply:

```
grep "boo" a_file
```

In this example, grep would loop through every line of the file "a_file" and print out every line that contains the word 'boo':

```
boot
book
booze
boots
```

Useful Options

This is nice, but if you were working with a large fortran file of something similar, it would probably be much more useful to you if the lines identified which line in the file they were, what way you could track down a particular string more easily, if you needed to open the file in an editor to make some changes. This can be accomplished by adding the -n parameter:

```
grep -n "boo" a_file
```

This yields a much more useful result, which explains which lines matched the search string:

```
1:boot
2:book
3:booze
5:boots
```

Another interesting switch is -v, which will print the negative result. In other words, grep will print all of the lines that do not match the search string, rather than printing the lines that match it. In the following case, grep will print every line that does not contain the string "boo," and will display the line numbers, as in the last example

```
grep -vn "boo" a_file
```

In this particular case, it will print

```
4:machine
6:bungie
7:bark
8:aaradvark
9:robots
```

The -c option tells grep to supress the printing of matching lines, and only display the number of lines that match the query. For instance, the following will print the number 4, because there are 4 occurences of "boo" in a_file.

```
grep -c "boo" a_file
4
```

The -l option prints only the filenames of files in the query that have lines that match the search string. This is useful if you are searching through multiple files for the same string. like so:

```
grep -l "boo" *
```

An option more useful for searching through non-code files is -i, ignore case. This option will treat upper and lower case as equivalent while matching the search string. In the following example, the lines containg "boo" will be printed out, even though the search string is uppercase.

```
grep -i "BOO" a_file
```

The -x option looks for eXact matches only. In other words, the following command will print nothing, because there are no lines that only contain the pattern "boo"

```
grep -x "boo" a_file
```

Finally, -A allows you to specify additional lines of context file, so you get the search string plus a number of additional lines, e.g.

```
grep -A2 “mach” a_file
machine
boots
bungie
```

Regular Expressions

A regular expression is a compact way of describing complex patterns in text. With grep, you can use them to search for patterns. Other tools let you use regular expressions (“regexps”) to modify the text in complex ways. The normal strings we have been using so far are in fact just very simple regular expressions. You may also come across them if you use wildcards such as '*' or '?' when listing filenames etc. You may use grep to search using basic regexps such as to search the file for lines ending with the letter e:

```
grep "e$" a_file
```

This will, of course, print

```
booze
machine
bungie
```

If you want a wider range of regular expression commands then you must use 'grep -E' (also known as the egrep command). For instance, the regexp command ? will match 1 or 0 occurences of the previous character:

```
grep -E "boots?" a_file
```

This query will return

```
boot
boots
```

You can also combine multiple searches using the pipe (|) which means 'or' so can do things like:

```
grep -E "boot|boots" a_file
boot
boots
```

Special characters

What if the thing you want to search for is a special character? If you wanted to find all lines containing the dollar character '$' then you cannot do grep '$' a_file as the '$' will be interpreted as a regexp and instead you will get all the lines which have anything as an end of line, ie all lines! The solution is to 'escape' the symbol, so you would use

```
grep '\$' a_file
broken$tuff
```

You can also use the '-F' option which stands for 'fixed string' or 'fast' in that it only searches for literal strings and not regexps.

More regexp examples

See http://gnosis.cx/publish/programming/regular_expressions.html

---

AWK

A text pattern scanning and processing language, created by Aho, Weinberger & Kernighan (hence the name). It can be quite sophisticated so this is NOT a complete guide, but should give you a taste of what awk can do. It can be very simple to use, and is strongly recommended. There are many on-line tutorials of varying complexity, and of course, there is always 'man awk'.

AWK basics

An awk program operates on each line of an input file. It can have an optional BEGIN{} section of commands that are done before processing any content of the file, then the main {} section works on each line of the file, and finally there is an optional END{} section of actions that happen after the file reading has finished:

```
BEGIN { …. initialization awk commands …}
{ …. awk commands for each line of the file…}
END { …. finalization awk commands …}
```

For each line of the input file, it sees if there are any pattern-matching instructions, in which case it only operates on lines that match that pattern, otherwise it operates on all lines. These 'pattern-matching' commands can contain regular expressions as for grep. The awk commands can do some quite sophisticated maths and string manipulations, and awk also supports associative arrays.

AWK sees each line as being made up of a number of fields, each being separated by a 'field separator'. By default, this is one or more space characters, so the line:

```
this is a line of text
```

contains 6 fields. Within awk, the first field is referred to as $1, the second as $2, etc. and the whole line is called $0. The field separator is set by the awk internal variable FS, so if you set FS=":" then it will divide a line up according to the position of the ':' which is useful for files like /etc/passwd etc. Other useful internal variables are NR which is the current record number (ie the line number of the input file) and NF which is the number of fields in the current line.

AWK can operate on any file, including std-in, in which case it is often used with the '|' command, for example, in combination with grep or other commands. For example, if I list all the files in a directory like this:

```
[mijp1@monty RandomNumbers]$ ls -l
total 2648
-rw------- 1 mijp1 mijp1 12817 Oct 22 00:13 normal_rand.agr
-rw------- 1 mijp1 mijp1  6948 Oct 22 00:17 random_numbers.f90
-rw------- 1 mijp1 mijp1 470428 Oct 21 11:56 uniform_rand_231.agr
-rw------- 1 mijp1 mijp1 385482 Oct 21 11:54 uniform_rand_232.agr
-rw------- 1 mijp1 mijp1 289936 Oct 21 11:59 uniform_rand_period_1.agr
-rw------- 1 mijp1 mijp1 255510 Oct 21 12:07 uniform_rand_period_2.agr
-rw------- 1 mijp1 mijp1 376196 Oct 21 12:07 uniform_rand_period_3.agr
```

```
-rw------- 1 mijp1 mijp1 494666 Oct 21 12:09 uniform_rand_period_4.agr
-rw------- 1 mijp1 mijp1 376286 Oct 21 12:05 uniform_rand_period.agr
```

I can see the file size is reported as the 5th column of data. So if I wanted to know the total size of all the files in this directory I could do:

```
[mijp1@monty RandomNumbers]$ ls -l | awk 'BEGIN {sum=0} {sum=sum+$5} END
{print sum}'
2668269
```

Note that 'print sum' prints the value of the variable sum, so if sum=2 then 'print sum' gives the output '2' whereas 'print $sum' will print '1' as the 2nd field contains the value '1'.

Hence it would be straightforwards to write an awk command that would calculate the mean and standard deviation of a column of numbers – you accumulate 'sum_x' and 'sum_x2' inside the main part, and then use the standard formulae to calculate mean and standard deviation in the END part.

AWK provides support for loops (both 'for' and 'while') and for branching (using 'if'). So if you wanted to trim a file and only operate on every 3rd line for instance, you could do this:

```
[mijp1@monty RandomNumbers]$ ls -l | awk '{for (i=1;i<3;i++) {getline};
print NR,$0}'
 3 -rw------- 1 mijp1 mijp1   6948 Oct 22 00:17 random_numbers.f90
 6 -rw------- 1 mijp1 mijp1 289936 Oct 21 11:59 uniform_rand_period_1.agr
 9 -rw------- 1 mijp1 mijp1 494666 Oct 21 12:09 uniform_rand_period_4.agr
10 -rw------- 1 mijp1 mijp1 376286 Oct 21 12:05 uniform_rand_period.agr
```

where the 'for' loop uses a 'getline' command to move through the file, and only prints out every 3rd line. Note that as the number of lines of the file is 10, which is not divisible by 3, the final command finishes early and so the final 'print $0' command prints line 10, which you can see as we also print out the line number using the NR variable.

AWK Pattern Matching

AWK is a line-oriented language. The pattern comes first, and then the action. Action statements are enclosed in { and }. Either the pattern may be missing, or the action may be missing, but, of course, not both. If the pattern is missing, the action is executed for every single record of input. A missing action prints the entire record.

AWK patterns include regular expressions (uses same syntax as 'grep -E') and combinations using the special symbols '&&' means 'logical AND', '||' means 'logical OR', '!' means 'logical NOT'. You can also do relational patterns, groups of patterns, ranges, etc.

AWK control statements include:

```
if (condition) statement [ else statement ]
while (condition) statement
do statement while (condition)
for (expr1; expr2; expr3) statement
for (var in array) statement
break
continue
```

```
exit [ expression ]
```

AWK input/output statements include:

close(file [, how])	Close file, pipe or co-process.
getline	Set $0 from next input record.
getline <file	Set $0 from next record of file.
getline var	Set var from next input record.
getline var <file	Set var from next record of file.
next	Stop processing the current input record. The next input record is read and processing starts over with the first pattern in the AWK program. If the end of the input data is reached, the END block(s), if any, are executed.
nextfile	Stop processing the current input file. If the end of the input data is reached, the END block(s), if any, are executed.
print	Prints the current record.
print expr-list	Prints expressions.
print expr-list >file	Prints expressions on file.
printf fmt, expr-list	Format and print.

NB The printf command lets you specify the output format more closely, using a C-like syntax, for example, you can specify an integer of given width, or a floating point number or a string, etc.

AWK numeric functions include:

atan2(y, x)	Returns the arctangent of y/x in radians.
cos(expr)	Returns the cosine of expr, which is in radians.
exp(expr)	The exponential function.
int(expr)	Truncates to integer.
log(expr)	The natural logarithm function.
Rand()	Returns a random number N, between 0 and 1, such that 0 <= N < 1.
sin(expr)	Returns the sine of expr, which is in radians.
sqrt(expr)	The square root function.
srand([expr])	Uses expr as a new seed for the random number generator. If no expr is provided, the time of day is used.

AWK string functions include:

`gsub(r, s [, t])`	For each substring matching the regular expression r in the string t, substitute the string s, and return the number of substitutions. If t is not supplied, use $0.
`index(s, t)`	Returns the index of the string t in the string s, or 0 if t is not present.
`length([s])`	Returns the length of the string s, or the length of $0 if s is not supplied.
`match(s, r [, a])`	Returns the position in s where the regular expression r occurs, or 0 if r is not present.
`split(s, a [, r])`	Splits the string s into the array a using the regular expression r, and returns the number of fields. If r is omitted, FS is used instead.
`sprintf(fmt, expr-list)`	Prints expr-list according to fmt, and returns the resulting string.
`strtonum(str)`	Examines str, and returns its numeric value.
`sub(r, s [, t])`	Just like gsub(), but only the first matching substring is replaced.
`substr(s, i [, n])`	Returns the at most n-character substring of s starting at i. If n is omitted, the rest of s is used.
`tolower(str)`	Returns a copy of the string str, with all the upper-case characters in str translated to their corresponding lower-case counterparts. Non-alphabetic characters are left unchanged.
`toupper(str)`	Returns a copy of the string str, with all the lower-case characters in str translated to their corresponding upper-case counterparts. Non-alphabetic characters are left unchanged.

AWK command-line and usage

You can pass variables into an awk program using the '-v' flag as many times as necessary, e.g.

```
awk -v skip=3 '{for (i=1;i<skip;i++) {getline}; print $0}' a_file
```

You can also write an awk program using an editor, and then save it as a special scripting file, e.g.

```
[mijp1@monty Comp_Lab]$ cat awk_strip
#!/usr/bin/awk -f
#only print out every 3rd line of input file

BEGIN {skip=3}

{for (i=1;i<skip;i++)
        {getline};
print $0}
```

which can then be used as a new additional command

```
[mijp1@monty Comp_Lab]$ chmod u+x awk_strip
[mijp1@monty Comp_Lab]$ ./awk_strip my_file.dat
```

sed = stream editor

sed performs basic text transformations on an input stream (a file or input from a pipeline) in a single pass through the stream, so it is very efficient. However, it is sed's ability to filter text in a pipeline which particularly distinguishes it from other types of editor.

SED basics

sed can be used at the command-line, or within a shell script, to edit a file non-interactively. Perhaps the most useful feature is to do a 'search-and-replace' for one string to another.

You can embed your sed commands into the command-line that invokes sed using the '-e' option, or put them in a separate file e.g. 'sed.in' and invoke sed using the '-f sed.in' option. This latter option is most used if the sed commands are complex and involve lots of regexps! For instance:

```
sed -e 's/input/output/' my_file
```

will echo every line from my_file to standard output, changing the first occurrence of 'input' on each line into 'output'. NB sed is line-oriented, so if you wish to change *every* occurrence on each line, then you need to make it a 'greedy' search & replace like so:

```
sed -e 's/input/output/g' my_file
```

The expression within the /.../ can be a literal string or a regexp.

NB By default the output is written to stdout. You may redirect this to a new file, or if you want to edit the existing file in place you should use the '-i' flag:

```
sed -e 's/input/output/' my_file > new_file
sed -i -e 's/input/output/' my_file
```

SED and regexps

What if one of the characters you wish to use in the search command is a special symbol, like '/' (e.g. in a filename) or '*' etc? Then you must escape the symbol just as for grep (and awk). Say you want to edit a shell scripts to refer to /usr/local/bin and not /bin any more, then you could do this

```
sed -e 's/\/bin/\/usr\/local\/bin/' my_script > new_script
```

What if you want to use a wildcard as part of your search – how do you write the output string? You need to use the special symbol '&' which corresponds to the pattern found. So say you want to take every line that starts with a number in your file and surround that number by parentheses:

```
sed -e 's/[0-9]*/(&)/' my_file
```

where [0-9] is a regexp range for all single digit numbers, and the '*' is a repeat count, means any number of digits.

You can also use positional instructions in your regexps, and even save part of the match in a pattern buffer to re-use elsewhere.

Other SED commands

The general form is

```
sed -e '/pattern/ command' my_file
```

where 'pattern' is a regexp and 'command' can be one of 's' = search & replace, or 'p' = print, or 'd' = delete, or 'i'=insert, or 'a'=append, etc. Note that the default action is to print all lines that do not match anyway, so if you want to suppress this you need to invoke sed with the '-n' flag and then you can use the 'p' command to control what is printed. So if you want to do a listing of all the sub-directories you could use

```
ls -l | sed -n -e '/^d/ p'
```

as the long-listing starts each line with the 'd' symbol if it is a directory, so this will only print out those lines that start with a 'd' symbol.

Similarly, if you wanted to delete all lines that start with the comment symbol '#' you could use

```
sed -e '/^#/ d' my_file
```

i.e. you can achieve the same effect in different ways!

You can also use the range form

```
sed -e '1,100 command' my_file
```

to execute 'command' on lines 1,100. You can also use the special line number '$' to mean 'end of file'. So if you wanted to delete all but the first 10 lines of a file, you could use

```
sed -e '11,$ d' my_file
```

You can also use a pattern-range form, where the first regexp defines the start of the range, and the second the stop. So for instance, if you wanted to print all the lines from 'boot' to 'machine' in the a_file example you could do this:

```
sed -n -e '/boot$/,/mach/p' a_file
```

which will then only print out (-n) those lines that are in the given range given by the regexps.

Further Reading

There is much more you can do with sed – see http://www.grymoire.com/Unix/Sed.html for a nice tutorial.