AWK Primer

Mark J. Duvall
mjduvall@hawaii.edu
UH Mānoa Department of Physics & Astronomy

This is a modification of an email I wrote for a colleague. I like AWK, but many people – even other physicists – seem unfamiliar with it, so I decided to write this up in case anyone finds it useful. Welcome to AWK!

Introduction

AWK is a simple but very fast and powerful language for processing text files. If you're familiar with `grep`, it's a lot like that, but a whole language! I've found it especially useful for text-formatted data and for code. Many of its functions are available in modern graphical applications; but these are often slow, resource-intensive, expensive, and worst of all, platform-specific. In contrast, AWK and text-based files are fast, lightweight, free / open-source, and best of all, *extremely portable*.

AWK's basic function is to test each line in the input file against a provided pattern and execute specified commands whenever a match is found.

A typical AWK program is therefore essentially just a series of IF-THEN statements:
IF /pattern/, THEN {commands}.

AWK sees files as "records," containing "fields." By default, each line of the input file is considered a record, and fields are separated by whitespace (i.e., spaces / tabs) within that record. Both these default delimiters can be changed if needed. AWK uses "$" to indicate a field number; $1 is the first field, $2 is the second field, etc.; and $0 means all fields – i.e., the entire line. For matching, AWK uses "∼" to mean "matches to", and the pattern is often provided as a regular expression. In my experience, having a basic knowledge of regular expressions can increase AWK's power & usefulness tremendously, but more on that later.

Basic Syntax: `awk '[matching test] {<commands>}' <filename>`

Common Variant: `cat <filename> | awk '[matching test] {<commands>}'`

Matching tests usually look something like: $0 ∼ /pattern/, which would check each line for any matches to your /pattern/.

The {`commands`} section is often something as simple as: {`print $1`}, which would print the first field.

To see how all of this comes together in the actual usage of AWK, see the example below.

∼ ∼ ∼

Basic Example

Suppose our input file is a list of people's heights called "heights.txt" and looks like this:

```
Alice 188cm
Bill 177cm
Clarice 175cm
```

It has 3 records, each having 2 fields. Suppose we want to find and print Alice's height; this means we want the first field to match her name. We want to tell AWK, "If the first field matches 'Alice,' then print the second field." That code would look like this:

```
user@host:~$ awk '$1 ~ /Alice/ {print $2}' heights.txt
188cm
```

Okay, so Alice is evidently 188 cm tall. Or, we could print the whole line by changing "print $2" to "print $0," just to make sure we got the right information:

```
user@host:~$ awk '$1 ~ /Alice/ {print $0}' heights.txt
Alice 188cm
```

So it worked! If you're wondering how this is any different from `grep`, the only real difference so far is that AWK is able to work field-by-field rather than line-by-line, which could be better than `grep` for particular file formats, but the real advantges of AWK haven't surfaced yet – keep reading!

—

A quick side note: We've enclosed "Alice" in slashes (in this particular case, double quotes would also have been fine) to indicate that it's supposed to be taken as a whole and not just a series of characters. Without the slashes, AWK would read 'Alice' as 'A *or* l *or* i *or* c *or* e,' and since all 3 names happen to contain an "i," we'd end up printing everyone's heights:

```
user@host:~$ awk '$1 ~ Alice {print $2}' heights.txt
188cm
177cm
175cm
```

This is obviously not what we wanted; that's why we need to use /Alice/ or "Alice". A tiny example of regular expressions ("regexes"): If we wanted to find Bill's height but there was also a "Billy" on the list, we'd be in trouble, since "Billy" contains a match for the pattern "Bill". We'd avoid this by using the regex special characters "^" for "beginning of string" and "$" for "end of string": the pattern /^Bill$/ would match only "Bill" and not "Billy".

—

That's it! Those are the basics of how AWK works. Once you get a bit used to it, you'll likely find yourself using it quite regularly if you work with text files a lot. Furthermore, this example only scratches the surface of what this language can do; for a *slightly* deeper demonstration of AWK's power, see the more advanced examples below. Happy AWKing!

# # # # #

Intermediate Example

Suppose we received a message from Clarice saying that there's been a mistake somewhere (not our fault, of course), and her height in our database is incorrect. Specifically, let's say her height is actually 185 cm, not 175 cm as listed in our file. No worries! One of AWK's most useful features is the ability to reassign new values to fields. In this case, we'll correct the data by processing the file in the following steps:

1. For any record(s) *not* matching "Clarice", print the whole record without modification.
2. For any record(s) matching "Clarice," replace the second field with the correct value, then print the corrected record.

We can accomplish this by giving AWK two sets of matching tests: one for "does not match *Clarice*," and one for "matches *Clarice*":

1. We use the "!" character to negate the match for *Clarice*:
   `$1 !~ /Clarice/ {print $0}`
   This means, "For any record whose name field does not contain a match for 'Clarice,' print the whole line."
2. We correct the height field by setting field $2 to 185cm:
   `$1 ~ /Clarice/ {$2=''185cm''; print $0}`
   This means, "Find a record whose name field matches 'Clarice,' then replace the height field with her corrected height and print the new record."

In this case, we'll want to write AWK's output to a new file, which we'll do via the usual shell output redirection "> *newfilename*" at the end of the command line. This would all be done at once, writing the output to *heights_corrected.txt*, as follows:

```
user@host:~$  cat heights.txt | awk '$1 !~ /Clarice/ {print $0}; \
> $1 ~ /Clarice/ {$2=''185cm''; print $0}' > heigts_corrected.txt
```

Checking our work (original file for reference):

```
user@host: $ cat heights.txt; echo; cat heights_corrected.txt
```

```
Alice 188cm
Bill 177cm
Clarice 175cm
```

```
Alice 188cm
Bill 177cm
Clarice 185cm
```

Success!

~ ~ ~

Advanced Example

A somewhat more advanced example of AWK usage would be to strip the "cm" from each line, perhaps to prepare heights.txt for importing into a spreadsheet or data analysis program. And while we're at it, let's make the file a little easier to read by putting a couple of tabs between the fields rather than just a space. I won't cover these details here, but just as a demonstration of AWK's power, it would look like this:

```
user@host:~$ cat heights.txt | awk '{match($2, /[[:digit:]]*/, a); \
> print $1"\t\t"a[0]}' > heights_without_units.txt
```

Now let's compare the original file with the new version:

```
user@host:~$ cat heights.txt; echo; cat heights_without_units.txt
```

```
Alice 188cm
Bill 177cm
Clarice 175cm

Alice		188
Bill		177
Clarice		175
```

There, that's much nicer. And since the heights in the new file are now pure numbers rather than character strings, we can operate on them as well! For example, we could print the names of all people shorter than 180 cm:

```
user@host:~$ awk '$2 < 180 {print $1}' heights_without_units.txt
Bill
Clarice
```

Just for fun, let's see what that entire process might look like in a single line using pipes. So, in a single line, we'll take the original input file, strip the "cm," then use the results to print the names of all people shorter than 180 cm:

```
user@host:~$ cat heights.txt | awk '{match($2, /[[:digit:]]*/, a); \
> print $1"\t\t"a[0]}' | awk '$2 < 180 {print $1}'
Bill
Clarice
```

Done! In a real-world scenario where we might have hundreds of names instead of just three, AWK has just made our lives *much, much easier*.

~ ~ ~

AWK Scripting

Finally, I'll note that you don't always have to enter all of your AWK code on the command line. In fact, as your AWK programs get more complex, you definitely won't want to do that! Fortunately, the AWK language can use scripts like most other languages. Just save your AWK code in a text file ('.awk' extension is common for convenience but not necessary), and use the '-f' option when you call AWK from the command line:
`awk -f <codefile> <targetfile>`

This also has the added benefit of syntax highlighting in compatible text editors. (*vim FTW!*)

For example, we could create a file called stripcm.awk that looks like this:

```
# stripcm.awk -- script to remove the ``cm'' from people's heights
{
match($2, /[[:digit:]]*/, a)
print $1``\t\t''a[0]
}
```

Then we would run it as follows:

```
user@host:~$ awk -f stripcm.awk heights.txt
```

```
Alice       188
Bill        177
Clarice     175
```

Units stripped & columns aligned!

Our entire process to 1) strip the "cm," turning the heights from strings into numbers, and 2) examine these numbers and print out the heights of everyone under 180 cm could now look like:

```
user@host:~$ cat heights.txt | awk -f stripcm.awk | awk '$2 < 180 print $1'
Bill
Clarice
```

That cleans things up quite a bit.

~ ~ ~

Bonus Example

Here's a real-world example from my last demonstration. Suppose we want to automate an operation on our newest data file using a BASH script. The first thing this script would need to do is identify the file we want, preferebly storing the filename into a BASH variable. Imagine our directory looks like this:

```
user@host:~/data$ ls
```

```
config.log  run3.dat
run1.dat    today.log
run2.dat    users.txt
```

We can use `ls *.dat` with options *long, time, reverse, human-readable* to get a detailed, time-ordered look at just our data files:

```
user@host:~/data$ ls -ltrh *.dat
```

```
-rw-rw-r-- 1 user user 253M Oct 22  2015 run1.dat
-rw-rw-r-- 1 user user 224M Mar 16 00:12 run2.dat
-rw-rw-r-- 1 user user 273M Jul  1 14:23 run3.dat
```

The time ordering guarantees that the last line will be the entry for our most recently-modified data file. We can quickly isolate this last line by piping the output of `ls` into the `tail` command with option *1*:

```
user@host:~/data$ ls -ltrh *.dat | tail -1
```

```
-rw-rw-r-- 1 user user 273M Jul  1 14:23 run3.dat
```

Now we can easily use AWK to get just the filename by piping this output into `awk` and printing the 9th field:

```
user@host:~/data$ ls -ltrh *.dat | tail -1 | awk '{print $9}'
```

```
run3.dat
```

Finally, to actually make this useful to our script, we want to store this output into a BASH variable using command substitution `$(...)` :

```
user@host:~/data$ newest_data_file=$(ls -ltrh *.dat | tail -1 | awk '{print $9}')
```

This is the line we would actually put in our script. This could be useful in a backup script, for example:

```
user@host:~/data$ newest_data_file=$(ls -ltrh /home/user/data/*.dat | tail -1 | awk '{print $9}')
user@host:~/data$ scp $newest_data_file user@remote-host:/data/backup/
```

```
run3.dat      100% 273M 1.0MB/s 04:33
```

Success! Automated backup of the newest data file – *in only two lines* – brought to you by BASH & AWK.

Disclaimer: I'm well aware that this particular case could've been handled by running `newest_data_file=$(ls -tr1 *.dat | tail -1)`*, but the point of this example was to illustrate AWK's ability to work in conjunction with other programs and easily handle a situation even if there isn't a handy command-line option to give you exactly what you need.*

# # # # #

And we're pau! There's a whole lot more to AWK than this – notably the BEGIN & END blocks and AWK's ability to use variables and arrays – but that's all for now. Enjoy!

Mark J. Duvall ~ mjduvall@hawaii.edu

# # # # #

See also:

My personal fork of RAT-PAC on GitHub, where you can search numerous examples of my own AWK usage:
`http://github.com/duvall3/rat-pac/tree/comparison/user/shell_scripts`

A wonderful guide to AWK, regular expressions, `sed`, and more:
`http://www.grymoire.com/Unix/Awk.html`

AWK on Wikipedia:
`http://en.wikipedia.org/wiki/AWK`