e-mail it!

Common threads: Awk by example, Part 1
An intro to the great language with the strange name

Daniel Robbins
President/CEO, Gentoo Technologies, Inc.
December 2000

Awk is a very nice language with a very strange name. In this first article of a three-part series, Daniel Robbins will quickly get your awk programming skills up to speed. As the series progresses, more advanced topics will be covered, culminating with an advanced real-world awk application demo.

In defense of awk

In this series of articles, I'm going to turn you into a proficient awk coder. I'll admit, awk doesn't have a very pretty or particularly "hip" name, and the GNU version of awk, called gawk, sounds downright weird. Those unfamiliar with the language may hear "awk" and think of a mess of code so backwards and antiquated that it's capable of driving even the most knowledgeable UNIX guru to the brink of insanity (causing him to repeatedly yelp "kill -9!" as he runs for coffee machine).

Sure, awk doesn't have a great name. But it is a great language. Awk is geared toward text processing and report generation, yet features many well-designed features that allow for serious programming. And, unlike some languages, awk's syntax is familiar, and borrows some of the best parts of languages like C, python, and bash (although, technically, awk was created before both python and bash). Awk is one of those languages that, once learned, will become a key part of your strategic coding arsenal.

The first awk

Let's go ahead and start playing around with awk to see how it works. At the command line, enter the following command:

```
$ awk '{ print }' /etc/passwd
```

You should see the contents of your /etc/passwd file appear before your eyes. Now, for an explanation of what awk did. When we called awk, we specified /etc/passwd as our input file. When we executed awk, it evaluated the print command for each line in /etc/passwd, in order. All output is sent to stdout, and we get a result identical to catting /etc/passwd.

Now, for an explanation of the `{ print }` code block. In awk, curly braces are used to group blocks of code together, similar to C. Inside our block of code, we have a single print command. In awk, when a print command appears by itself, the full contents of the current line are printed.

Here is another awk example that does exactly the same thing:

```
$ awk '{ print $0 }' /etc/passwd
```

In awk, the `$0` variable represents the entire current line, so `print` and `print $0` do exactly the same thing.

If you'd like, you can create an awk program that will output data totally unrelated to the input data. Here's an example:

```
$ awk '{ print "" }' /etc/passwd
```

Whenever you pass the "" string to the print command, it prints a blank line. If you test this script, you'll find that awk outputs one blank line for every line in your /etc/passwd file. Again, this is because awk executes your script for every line in the input file. Here's another example:

```
$ awk '{ print "hiya" }' /etc/passwd
```

Running this script will fill your screen with hiya's. :)

Multiple fields
Awk is really good at handling text that has been broken into multiple logical fields, and allows you to effortlessly reference each individual field from inside your awk script. The following script will print out a list of all user accounts on your system:

```
$ awk -F":" '{ print $1 }' /etc/passwd
```

Above, when we called awk, we use the -F option to specify ":" as the field separator. When awk processes the `print $1` command, it will print out the first field that appears on each line in the input file. Here's another example:

```
$ awk -F":" '{ print $1 $3 }' /etc/passwd
```

Here's an excerpt of the output from this script:

```
halt7
operator11
root0
shutdown6
sync5
bin1
....etc.
```

As you can see, awk prints out the first and third fields of the /etc/passwd file, which happen to be the username and uid fields respectively. Now, while the script did work, it's not perfect -- there aren't any spaces between the two output fields! If you're used to programming in bash or python, you may have expected the `print $1 $3` command to insert a space between the two fields. However, when two strings appear next to each other in an awk program, awk concatenates them without adding an intermediate space. The following command will insert a space between both fields:

```
$ awk -F":" '{ print $1 " " $3 }' /etc/passwd
```

When you call print this way, it'll concatenate `$1`, " ", and `$3`, creating readable output. Of course, we can also insert some text labels if needed:

```
$ awk -F":" '{ print "username: " $1 "\t\tuid:" $3" }' /etc/passwd
```

This will cause the output to be:

```
username: halt          uid:7
username: operator      uid:11
username: root          uid:0
username: shutdown      uid:6
username: sync          uid:5
username: bin           uid:1
....etc.
```

External scripts
Passing your scripts to awk as a command line argument can be very handy for small one-liners, but when it comes to complex, multi-line programs, you'll definitely want to compose your script in an external file. Awk can then be told to source this script file by passing it the -f option:

```
$ awk -f myscript.awk myfile.in
```

Putting your scripts in their own text files also allows you to take advantage of additional awk features. For example, this multi-line script does the same thing as one of our earlier one-liners, printing out the first field of each line in /etc/passwd:

```
BEGIN {
        FS=":"
}

{ print $1 }
```

The difference between these two methods has to do with how we set the field separator. In this script, the field separator is specified within the code itself (by setting the FS variable), while our previous example set FS by passing the -F":" option to awk on the command line. It's generally best to set the field separator inside the script itself, simply because it means you have one less command line argument to remember to type. We'll cover the FS variable in more detail later in this article.

The BEGIN and END blocks
Normally, awk executes each block of your script's code once for each input line. However, there are many programming situations where you may need to execute initialization code *before* awk begins processing the text from the input file. For such situations, awk allows you to define a BEGIN block. We used a BEGIN block in the previous example. Because the BEGIN block is evaluated before awk starts processing the input file, it's an excellent place to initialize the FS (field separator) variable, print a heading, or initialize other global variables that you'll reference later in the program.

Awk also provides another special block, called the END block. Awk executes this block after all lines in the input file have been processed. Typically, the END block is used to perform final calculations or print summaries that should appear at the end of the output stream.

Regular expressions and blocks
Awk allows the use of regular expressions to selectively execute an individual block of code, depending on whether or not the regular expression matches the current line. Here's an example script that outputs only those lines that contain the character sequence `foo`:

```
/foo/ { print }
```

Of course, you can use more complicated regular expressions. Here's a script that will print only lines that contain a floating point number:

```
/[0-9]+\.[0-9]*/ { print }
```

Expressions and blocks
There are many other ways to selectively execute a block of code. We can place any kind of boolean expression before a code block to control when a particular block is executed. Awk will execute a code block only if the preceding boolean expression evaluates to true. The following example script will output the third field of all lines that have a first field equal to `fred`. If the first field of the current line is not equal to `fred`, awk will continue processing the file and will not execute the `print` statement for the current line:

```
$1 == "fred" { print $3 }
```

Awk offers a full selection of comparison operators, including the usual "==", "<", ">", "<=", ">=", and "!=". In addition, awk provides the "~" and "!~" operators, which mean "matches" and "does not match". They're used by specifying a variable on the left side of the operator, and a regular expression on the right side. Here's an example that will print only the third field on the line if the fifth field on the same line contains the character sequence `root`:

```
$5 ~ /root/ { print $3 }
```

Conditional statements
Awk also offers very nice C-like `if` statements. If you'd like, you could rewrite the previous script using an `if` statement:

```
{
        if ( $5 ~ /root/ ) {
                print $3
        }
}
```

Both scripts function identically. In the first example, the boolean expression is placed outside the block, while in the second example, the block is executed for every input line, and we selectively perform the print command by using an `if` statement. Both methods are available, and you can choose the one that best meshes with the other parts of your script.

Here's a more complicated example of an awk `if` statement. As you can see, even with complex, nested conditionals, `if` statements look identical to their C counterparts:

```
{
        if ( $1 == "foo" ) {
                if ( $2 == "foo" ) {
                        print "uno"
                } else {
                        print "one"
                }
        } else if ($1 == "bar" ) {
                print "two"
        } else {
                print "three"
        }
}
```

Using `if` statements, we can also transform this code:

```
! /matchme/ { print $1 $3 $4 }
```

to this:

```
{
        if ( $0 !~ /matchme/ ) {
                print $1 $3 $4
        }
}
```

Both scripts will output only those lines that *don't* contain a `matchme` character sequence. Again, you can choose the method that works best for your code. They both do the same thing.

Awk also allows the use of boolean operators "||" (for "logical or") and "&&"(for "logical and") to allow the creation of more complex boolean expressions:

```
( $1 == "foo" ) && ( $2 == "bar" ) { print }
```

This example will print only those lines where field one equals `foo` *and* field two equals `bar`.

Numeric variables!
So far, we've either printed strings, the entire line, or specific fields. However, awk also allows us to perform both integer and floating point math. Using mathematical expressions, it's very easy to write a script that counts the number of blank lines in a file. Here's one that does just that:

```
BEGIN   { x=0 }
/^$/    { x=x+1 }
END     { print "I found " x " blank lines. :)" }
```

In the BEGIN block, we initialize our integer variable `x` to zero. Then, each time awk encounters a blank line, awk will execute the `x=x+1` statement, incrementing `x`. After all the lines have been processed, the END block will execute, and awk will print out a final summary, specifying the number of blank lines it found.

Stringy variables
One of the neat things about awk variables is that they are "simple and stringy." I consider awk variables "stringy" because all awk variables are stored internally as strings. At the same time, awk variables are "simple" because you can perform mathematical operations on a variable, and as long as it contains a valid numeric string, awk automatically takes care of the string-to-number conversion steps. To see what I mean, check out this example:

```
x="1.01"
# We just set x to contain the *string* "1.01"
x=x+1
# We just added one to a *string*
print x
# Incidentally, these are comments :)
```

Awk will output:

```
2.01
```

Interesting! Although we assigned the string value `1.01` to the variable `x`, we were still able to add one to it. We wouldn't be able to do this in bash or python. First of all, bash doesn't support floating point arithmetic. And, while bash has "stringy" variables, they aren't "simple"; to perform any mathematical operations, bash requires that we enclose our math in an ugly `$( )` construct. If we were using python, we would have to explicitly convert our `1.01` string to a floating point value before performing any arithmetic on it. While this isn't difficult, it's still an additional step. With awk, it's all automatic, and that makes our code nice and clean. If we wanted to square and add one to the first field in each input line, we would use this script:

```
{ print ($1^2)+1 }
```

If you do a little experimenting, you'll find that if a particular variable doesn't contain a valid number, awk will treat that variable as a numerical zero when it evaluates your mathematical expression.

Lots of operators
Another nice thing about awk is its full complement of mathematical operators. In addition to standard addition, subtraction, multiplication, and division, awk allows us to use the previously demonstrated exponent operator "^", the modulo (remainder) operator "%", and a bunch of other handy assignment operators borrowed from C.

These include pre- and post-increment/decrement (`i++`, `--foo`), add/sub/mult/div assign operators (`a+=3`, `b*=2`, `c/=2.2`, `d-=6.2`). But that's not all -- we also get handy modulo/exponent assign ops as well (`a^=2`, `b%=4`).

Field separators
Awk has its own complement of special variables. Some of them allow you to fine-tune how awk functions, while others can be read to glean valuable information about the input. We've already touched on one of these special variables, FS. As mentioned earlier, this variable allows you to set the character sequence that awk expects to find between fields. When we were using /etc/passwd as input, FS was set to ":". While this did the trick, FS allows us even more flexibility.

The FS value is not limited to a single character; it can also be set to a regular expression, specifying a character pattern of any length. If you're processing fields separated by one or more tabs, you'll want to set FS like so:

```
FS="\t+"
```

Above, we use the special "+" regular expression character, which means "one or more of the previous character".

If your fields are separated by whitespace (one or more spaces or tabs), you may be tempted to set FS to the following regular expression:

```
FS="[[:space:]+]"
```

While this assignment will do the trick, it's not necessary. Why? Because by default, FS is set to a single space character, which awk interprets to mean "one or more spaces or tabs." In this particular example, the default FS setting was exactly what you wanted in the first place!

Complex regular expressions are no problem. Even if your records are separated by the word "foo," followed by three digits, the following regular expression will allow your data to be parsed properly:

```
FS="foo[0-9][0-9][0-9]"
```

Number of fields
The next two variables we're going to cover are not normally intended to be written to, but are normally read and used to gain useful information about the input. The first is the NF variable, also called the "number of fields" variable. Awk will automatically set this variable to the number of fields in the current record. You can use the NF variable to display only certain input lines:

```
NF == 3 { print "this particular record has three fields: " $0 }
```

Of course, you can also use the NF variable in conditional statements, as follows:

```
{
        if ( NF > 2 ) {
                print $1 " " $2 ":" $3
        }
}
```

Record number
The record number (NR) is another handy variable. It will always contain the number of the current record (awk counts the first record as record number 1). Up until now, we've been dealing with input files that contain one record per line. For these situations, NR will also tell you the current line number. However, when we start to process multi-line records later in the series, this will no longer be the case, so be careful! NR can be used like the NF variable to print only certain lines of the input:

```
(NR < 10 ) || (NR > 100) { print "We are on record number 1-9 or 101+" }
```

Another example:

```
{
        #skip header
        if ( NR > 10 ) {
                print "ok, now for the real information!"
        }
}
```

Awk provides additional variables that can be used for a variety of purposes. We'll cover more of these variables in later articles.

We've come to the end of our initial exploration of awk. As the series continues, I'll demonstrate more advanced awk functionality, and we'll end the series with a real-world awk application. In the meantime, if you're eager to learn more, check out the resources listed below.

Resources

- If you'd like a good old-fashioned book, O'Reilly's sed & awk, 2nd Edition is a wonderful choice.
- Be sure to check out the comp.lang.awk FAQ. It also contains lots of additional awk links.
- Patrick Hartigan's awk tutorial is packed with handy awk scripts.
- Thompson's TAWK Compiler compiles awk scripts into fast binary executables. Versions are available for Windows, OS/2, DOS, and UNIX.
- The GNU Awk User's Guide is available for online reference.

About the author

Residing in Albuquerque, New Mexico, Daniel Robbins is the President/CEO of Gentoo Technologies, Inc., the creator of **Gentoo Linux**, an advanced Linux for the PC, and the **Portage** system, a next-generation ports system for Linux. He has also served as a contributing author for the Macmillan books *Caldera OpenLinux Unleashed*, *SuSE Linux Unleashed*, and *Samba Unleashed*. Daniel has been involved with computers in some fashion since the second grade, when he was first exposed to the Logo programming language as well as a potentially dangerous dose of Pac Man. This probably explains why he has since served as a Lead Graphic Artist at **SONY Electronic Publishing/Psygnosis**. Daniel enjoys spending time with his wife, Mary, and his new baby daughter, Hadassah. You can reach Daniel at drobbins@gentoo.org.

e-mail it!

What do you think of this article?

○ Killer! (5) ○ Good stuff (4) ○ So-so; not bad (3) ○ Needs work (2) ○ Lame! (1)

Comments?

Submit feedback