

## Getting started with awk

This gref is written for a semi-knowledgable UNIX user who has just come up against a problem and has been advised to use *awk* to solve it. Perhaps one of the examples can be quickly modified for immediate use.

---

- **Pre-Info**
    - [Ohhh ohh what flavor?](#)
    - [For More Info](#)
    - [References](#)
  - [Introduction](#)
  - [The Basics](#)
  - [Some Samples](#)
    - [Whats a pattern, whats an action?](#)
    - [With a regular expression?](#)
    - [Comparisons](#)
    - [Negate Operator](#)
    - [Booleans](#)
    - [start and end](#)
    - [BEGIN and END](#)
    - [Multiple pattern action pairs](#)
    - [Awk variables](#)
    - [Awk for while do](#)
    - [Awk arrays](#)
    - [Awk from a file](#)
    - [Awk to create C code](#)
  - [Awk punctuation guide](#)
  - [A large awk example](#)
- 

## Ohhh ohh what flavor?

There are three popular versions of awk: *awk*, *nawk*, and *gawk*. The last two are compatible with the first one.

---

## For More Info

If you are looking for a more in-depth online document about awk, you should read the "info" pages for gawk. The info pages contain both the official documentation for gawk and a good introduction to gawk. To read the info pages, type

```
>info gawk
```

at a shell prompt. It is also possible to print out the info pages in a nice-looking format. Send email to [help@cs.hmc.edu](mailto:help@cs.hmc.edu) and ask how to do that.

---

## References

A good reference for *awk* is the O'Reilly handbook for *sed and awk*. There should be a copy available in the CS Department library. Further references are the *UNIX in a Nutshell* and *UNIX Power Tools* books, also in the CS Department library. The *Power Tools* book seems to quote quite a bit from the *Sed and Awk* book, though. Once you know a bit about *awk*, the man pages become more useful.

## The Awk Newsgroup

The newsgroup for *awk* is [comp.lang.awk](http://comp.lang.awk)

---

## Introduction

- *awk* reads from a file or from its standard input, and outputs to its standard output. You will generally want to redirect that into a file, but that is not done in these examples just because it takes up space. *awk* does not get along with non-text files, like executables and FrameMaker files. If you need to edit those, use a binary editor like hexl-mode in emacs.
- The most frustrating thing about trying to learn *awk* is getting your program past the shell's parser. The proper way is to use single quotes around the program, like so:

```
>awk '{print $0}' filename
```

The single quotes protect almost everything from the shell. In *csh* or *tcsh*, you still have to watch out for exclamation marks, but other than that, you're safe.

- The second most frustrating thing about trying to learn *awk* is the lovely error messages:
  - `awk '{print $0,}' filename`
  - `awk: syntax error near line 1`
  - `awk: illegal statement near line 1`

*gawk* generally has better error messages. At least it tells you where in the line something went wrong:

```
gawk '{print $0,}' filename
gawk: cmd. line:1: {print $0,}
gawk: cmd. line:1:                ^ parse error
```

So, if you're having problems getting *awk* syntax correct, switch to *gawk* for a while.

---

## Some basics:

- Awk recognizes the concepts of "file", "record", and "field".
- A file consists of records, which by default are the lines of the file. One line becomes one record.
- Awk operates on one record at a time.
- A record consists of fields, which by default are separated by any number of spaces or tabs.
- Field number 1 is accessed with \$1, field 2 with \$2, and so forth. \$0 refers to the whole record.

---

## Some Samples

Perhaps the quickest way of learning awk is to look at some sample programs. The one above will print the file in its entirety, just like *cat*(1). Here are some others, along with a quick description of what they do.

```
>awk '{print $2,$1}' filename
```

will print the second field, then the first. All other fields are ignored.

```
>awk '{print $1,$2,sin($3/$2)}' filename
```

will print the first and second fields, and then the sine of the third field divided by the second. So, the second and third field had better be numbers. Awk has other built in math functions like sine; read the manpage to see which ones.

"I still say awk '{print \$1}' a lot."  
*the inventor of PERL, Larry Wall (lwall@netlabs.com)*

What if you don't want to apply the program to each line of the file? Say, for example, that you only wanted to process lines that had the first field greater than the second. The following program will do that:

```
>awk '$1 > $2 {print $1,$2,$1-$2}' filename
```

The part outside the curly braces is called the "pattern", and the part inside is the "action". The comparison operators include the ones from C:

```
== != < > <= >= ?:
```

If no pattern is given, then the action applies to all lines. This fact was used in the sample programs above. If no action is given, then the entire line is printed. If "print" is used all by itself, the entire line is printed. Thus, the following are equivalent:

```
awk '$1 > $2' filename
awk '$1 > $2{print}' filename
awk '$1 > $2{print $0}' filename
```

The various fields in a line can also be treated as strings instead of numbers. To compare a field to a string, use the following method:

```
>awk '$1=="foo"{print $2}' filename
```

---

## Using regular expressions

What if you want lines in which a certain string is found? Just put a regular expression (in the manner of *egrep*(1) ) into the pattern, like so:

```
>awk '/foo.*bar/{print $1,$3}' filename
```

This will print all lines containing the word "foo" and then later the word "bar". If you want only those lines where "foo" occurs in the second field, use the ~ ("contains") operator:

```
>awk '$2~/foo/{print $3,$1}' filename
```

If you want lines where "foo" does not occur in the second field, use the negated ~ operator, !~

```
>awk '$2!~/foo/{print $3,$1}' filename
```

This operator can be read as "does not contain".

---

## Booleans

You can produce complicated patterns with the boolean operators from C, which are ! for "not", && for "and", and || for "or". Parentheses can be used for grouping.

---

## Start and End

There are three special forms of patterns that do not fit the above descriptions. One is the start-end pair of regular expressions. For example, to print all lines between and including lines that contained "foo" and "bar", you would use

```
>awk '/foo/,/bar/' filename
```

---

## Begin and End

The other two special forms are similar; they are the BEGIN and END patterns. Any action associated with the BEGIN pattern will happen before any line-by-line processing is done. Actions with the END pattern will happen after all lines are processed.

But how do you put more than one pattern-action pair into an awk program? There are several choices.

1. One is to just mash them together, like so:

```
>
```

```
awk 'BEGIN{print"fee"} $1=="foo"{print"fi"}
    END{print"fo fum"}' filename
```

2. Another choice is to put the program into a file, like so:

```
3.     BEGIN{print"fee"}
4.     $1=="foo"{print"fi"}
5.     END{print"fo fum"}
```

Let's say that's in the file *giant.awk*. Now, run it using the "-f" flag to awk:

```
>awk -f giant.awk filename
```

6. A third choice is to create a file that calls awk all by itself. The following form will do the trick:

```
7.     #!/usr/bin/awk -f
8.     BEGIN{print"fee"}
9.     $1=="foo"{print"fi"}
10.    END{print"fo fum"}
```

If we call this file *giant2.awk*, we can run it by first giving it execute permissions,

```
>chmod u+x giant2.awk
```

and then just call it like so:

```
>./giant2.awk filename
```

awk has variables that can be either real numbers or strings. For example, the following code prints a running total of the fifth column:

```
>awk '{print x+=$5,$0 }' filename
```

This can be used when looking at file sizes from an "ls -l". It is also useful for balancing one's checkbook, if the amount of the check is kept in one column.

---

## Awk variables

awk variables are initialized to either zero or the empty string the first time they are used. Which one depends on how they are used, of course.

Variables are also useful for keeping intermediate values. This example also introduces the use of

semicolons for separating statements:

```
>awk '{d=($2-($1-4));s=($2+$1);print d/sqrt(s),d*d/s }' filename
```

Note that the final statement, a "print" in this case, does not need a semicolon. It doesn't hurt to put it in, though.

- Integer variables can be used to refer to fields. If one field contains information about which other field is important, this script will print only the important field:

```
>awk '{imp=$1; print $imp }' filename
```

- The special variable *NF* tells you how many fields are in this record. This script prints the first and last field from each record, regardless of how many fields there are:

```
>awk '{print $1,$NF }' filename
```

- The special variable *NR* tells you which record this is. It is incremented each time a new record is read in. This gives a simple way of adding line numbers to a file:

```
>awk '{print NR,$0 }' filename
```

Of course, there are a myriad of other ways to put line numbers on a file using the various UNIX utilities. This is left as an exercise for the reader.

- The special variable *FS* (Field Separator) determines how awk will split up each record into fields. This variable can be set on the command line. For example, */etc/passwd* has its fields separated by colons.

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

This variable can actually be set to any regular expression, in the manner of *egrep(1)*.

The various fields are also variables, and you can assign things to them. If you wanted to delete the 10th field from each line, you could do it by printing fields 1 through 9, and then from 11 on using a for-loop (see below). But, this will do it very easily:

```
>awk '{$10=""; print }' filename
```

In many ways, awk is like C. The "for", "while", "do-while", and "if" constructs all exist. Statements can be grouped with curly braces. This script will print each field of each record on its own line.

```
>awk '{for(i=1;i<=NF;i++) print $i }' filename
```

If you want to produce format that is a little better formatted than the "print" statement gives you, you can use "printf" just like in C. Here is an example that treats the first field as a string, and then does some numeric stuff

```
>awk '{printf("%s %03d %02d %.15g\n",$1,$2,$3,$3/$2); }' filename
```

Note that with printf, you need the explicit newline character.

We can use "printf" to print stuff without the newline, which is useful in a for loop. This script prints each record with each of its fields reversed. Ok, so it isn't very useful.

```
>awk '{for(i=NF;i > 0;i--) printf("%s",$i); printf("\n"); }'
filename
```

---

## Awk Arrays

awk has arrays, but they are only indexed by strings. This can be very useful, but it can also be annoying. For example, we can count the frequency of words in a document (ignoring the icky part about printing them out):

```
>awk '{for(i=1;i <=NF;i++) freq[$i]++ }' filename
```

The array will hold an integer value for each word that occurred in the file. Unfortunately, this treats "foo", "Foo", and "foo," as different words. Oh well. How do we print out these frequencies? awk has a special "for" construct that loops over the values in an array. This script is longer than most command lines, so it will be expressed as an executable script:

```
#!/usr/bin/awk -f
{for(i=1;i <=NF;i++) freq[$i]++ }
END{for(word in freq) print word, freq[word] }
```

This loop-over-an-array seems to go in no particular order. Thus, the output from a program like this must often be piped through *sort(1)* in order to be useful.

Multi-dimensional arrays are implemented in an odd way. The given indices are concatenated together (with a special separator) to get one string, and it is used as the index. This program will print the word-pair frequencies:

```
#!/usr/bin/awk -f
{for(i=1;i < NF;i++) freq[$i,$(i+1)]++ }
END{for(words in freq) print words, freq[words] }
```

Unfortunately, this will print out the separator, which is by default not a common character. You can change this by assigning something logical like a space to the variable SUBSEP using *nawk* or *gawk* (it's not allowed in plain awk).

```
#!/usr/bin/awk -f
BEGIN{SUBSEP=" "}
{for(i=1;i < NF;i++) freq[$i,$(i+1)]++ }
END{for(words in freq) print words, freq[words] }
```

Unfortunately (that word seems to occur a lot when talking about awk arrays), this doesn't let you refer to the indices individually. The secret to this is to use the "split" function, which breaks a string up into an array.

```
#!/usr/bin/awk -f
BEGIN{SUBSEP=" "}
{for(i=1;i < NF;i++) freq[$i,$(i+1)]++ }
```

```

END{ for(words in freq)
    {
    split(words,word,SUBSEP);
    print word[1], freq[words],word[2];
    }
}

```

---

When you're using an awk script in a file, you can break your program across multiple lines to make it easier to read. Comments are started the same way as in *sh* programming, with a #

```

#!/usr/bin/awk -f
# this program prints the frequencies of word pairs
BEGIN{SUBSEP=""} # set the index separator
                # to a nice character
{for(i=1;i < NF;i++) freq[$i,$(i+1)]++}
END{ for(words in freq)
    {
    # just to show we can put a comment in here.
    split(words,word,SUBSEP); # or here
    print word[1], freq[words],word[2];
    }
}

```

You can use awk to create text, as well as just process existing text. It is useful for quickly generating tables of function values, without the hassle of compiling a C program. For example, it can show that  $\sin(x)/x$  approaches 1 as  $x$  approaches zero:

```
>awk '{x=1.0/NR; print x, sin(x)/x;}'
```

will print a new value each time it reads a new line. So, you can hit return until you have all the values you need. Alternately, if you need a set number of values, you can do

```
>awk 'BEGIN{for(i=1;i <=30;i++){x=1.0/i;print x, sin(x)/x;}}'
/dev/null
```

where 30 is the set number of values.

It seems twisted\*, but awk can be used to generate C code that one doesn't want to type by hand. For example, this script will generate an explicit 3x3 matrix multiplication routine:

```

gawk 'BEGIN{
    for(i=0;i<3;i++)
    for(j=0;j<3;j++){
    printf("d[%d][%d]=",i,j);
    for(k=0;k<3;k++){
    printf("l[%d][%d]*r[%d][%d]s",
        i,k,k,j,(k<2)?"+":";\n");
    }
    }
}'

```

\* *ok, maybe it is twisted.*

---

## Punctuation guide:

{ }	used around the action, and to group statements in the action.
\$	denotes a field. \$1 is the first field, \$0 is the whole record.
~	the "contains" operator. "foobar"~"foo" is true. Strings only.
!~	the "does not contain" operator. Strings only.
==	the equality operator. Works for numbers or strings
< > <= >= !=	inequality operators. Work for numbers or strings.
#	the begin-comment character
,	separates things in a "print" or "printf" statement.
;	separates statements.
//	used around a regular expression
&&	Boolean and
	Boolean or
!	boolean not
()	used for grouping Boolean expressions, passing arguments to functions, and around conditions for "for","while", etc.

---

## And now for a grand example:

```
# This awk program collects statistics on two
# "random variables" and the relationships
# between them. It looks only at fields 1 and
# 2 by default Define the variables F and G
# on the command line to force it to look at
# different fields. For example:
# awk -f stat_2o1.awk F=2 G=3 stuff.dat \
# F=3 G=5 otherstuff.dat
# or, from standard input:
# awk -f stat_2o1.awk F=1 G=3
# It ignores blank lines, lines where either
# one of the requested fields is empty, and
# lines whose first field contains a number
# sign. It requires only one pass through the
# data. This script works with vanilla awk
# under SunOS 4.1.3.
```

```

BEGIN{
    F=1;
    G=2;
}
length($F) > 0 && \
length($G) > 0 && \
$1 !~/^#/ {
    sx1+= $F; sx2 += $F*$F;
    syl+= $G; sy2 += $G*$G;
    sxy1+= $F*$G;
    if( N==0 ) xmax = xmin = $F;
    if( xmin > $F ) xmin=$F;
    if( xmax < $F ) xmax=$F;
    if( N==0 ) ymax = ymin = $G;
    if( ymin > $G ) ymin=$G;
    if( ymax < $G ) ymax=$G;
    N++;
}

END {
    printf("%d # N\n" ,N );
    if ( N <= 1)
    {
        printf("What's the point?\n");
        exit 1;
    }
    printf("%g # xmin\n",xmin);
    printf("%g # xmax\n",xmax);
    printf("%g # xmean\n",xmean=sx1/N);
    xSigma = sx2 - 2 * xmean * sx1+ N*xmean*xmean;
    printf("%g # xvar\n" ,xvar =xSigma/ N );
    printf("%g # xvar unbiased\n",xvaru=xSigma/(N-1));
    printf("%g # xstddev\n" ,sqrt(xvar ));
    printf("%g # xstddev unbiased\n",sqrt(xvaru));

    printf("%g # ymin\n",ymin);
    printf("%g # ymax\n",ymax);
    printf("%g # ymean\n",ymean=syl/N);
    ySigma = sy2 - 2 * ymean * syl+ N*ymean*ymean;
    printf("%g # yvar\n" ,yvar =ySigma/ N );
    printf("%g # yvar unbiased\n",yvaru=ySigma/(N-1));
    printf("%g # ystddev\n" ,sqrt(yvar ));
    printf("%g # ystddev unbiased\n",sqrt(yvaru));
    if ( xSigma * ySigma <= 0 )
        r=0;
    else
        r=(sxy1 - xmean*syl- ymean * sx1+ N * xmean * ymean)
        /sqrt(xSigma * ySigma);
    printf("%g # correlation coefficient\n", r);
    if( r > 1 || r < -1 )
        printf("SERIOUS ERROR! CORRELATION COEFFICIENT");
        printf(" OUTSIDE RANGE -1..1\n");

    if( 1-r*r != 0 )
        printf("%g # Student's T (use with N-2 degfreed)\n&", \
            t=r*sqrt((N-2)/(1-r*r)) );
    else
        printf("0 # Correlation is perfect,");
        printf(" Student's T is plus infinity\n");
    b = (sxy1 - ymean * sx1)/(sx2 - xmean * sx1);
    a = ymean - b * xmean;
}

```

```

ss=sy2 - 2*a*sy1- 2*b*sx1 + N*a*a + 2*a*b*sx1+ b*b*sx2 ;
ss/= N-2;
printf("%g # a = y-intercept\n", a);
printf("%g # b = slope\n", b);
printf("%g # s^2 = unbiased estimator for sigsq\n",ss);
printf("%g + %g * x # equation ready for cut-and-paste\n",a,b);
ra = sqrt(ss * sx2 / (N * xSigma));
rb = sqrt(ss / (xSigma));
printf("%g # radius of confidence interval ");
printf("for a, multiply by t\n",ra);
printf("%g # radius of confidence interval ");
printf("for b, multiply by t\n",rb);
}

```

This documentation was originally written by Andrew M. Ross.

---

Copyright (c) HMC Computer Science Department. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled "[GNU Free Documentation License](#)."