

AWK

Q: Where did the term "AWK" come from?

A: A, W, and K are the last name initials of the three Bell Labs scientists that created the language...Aho, Weinberger, and Kernighan

Q: What does AWK do?

A: AWK is a pattern matching system that includes and surpasses the power of grep and sed with regard to text manipulation. Note that AWK is not limited to text manipulation...AWK can handle just about anything thrown at it.

Structure of an AWK Program

Each AWK program is a sequence of one or more pattern-action statements with the syntax:

```
pattern { action }  
pattern { action }
```

...

AWK scans a sequence of input lines searching for lines that are *matched* by any patterns in the program

Every line in turn is tested against each of the patterns, and for every match, the action(s) will be performed until input is exhausted

How AWK Works

Given the data file named `empfile.dat` containing the following:

Beth	4.00	0
Dan	3.75	0
Kathy	4.00	10
Mark	5.00	20
Mary	5.50	22
Susie	4.25	18

where *column 1* is employee name, *column 2* is rate of pay, and *column 3* is hours worked

Here are some examples:

```
$ awk '$3 > 0 { print $1, $2, $3 }' empfile.dat
```

will try to find all lines in the `empfile.dat` file where the *value in column 3 is greater than 0*.

If the expression `$3 > 0` is *true* for any line, the action requires that the contents of fields (columns) 1, 2, and 3 be printed on the screen

NOTE: the comma between the column numbers in the action statement implies that a space will be generated between the fields in the print-out

NOTE: there is a space after the opening curly brace, and a space before the closing curly brace

So, the AWK command:

```
$ awk '$3 > 0 { print $1, $2, $3 }' empfile.dat
```

returns

Kathy	4.00	10
Mark	5.00	20
Mary	5.50	22
Susie	4.25	18

Another example

```
$ awk '$3 == 0 { print $1 }' empfile.dat
```

will return

Beth
Dan

NOTE:

The string contained between the single-quote marks is also called the *program*

Alternative ways of calling AWK

```
$ awk 'program' input files
```

attempts to apply the program to as many files that are placed as arguments to AWK

Alternative ways of calling AWK (continued)

Example of scanning multiple files:

```
awk '$3 > 0 { print $1 }' file1, file2
```

will print the contents of the first column of every line in file1 and file2 where the pattern/argument is true for the value in the third column

Example of pattern with no given action:

```
awk '$3 == 0' file1
```

by default will print the contents of every column in each line in file1 where *\$3 == 0 is true*

Example of action with no given pattern:

```
awk '{ print $1 }' file1
```

by default will print the contents of the first column in each line in file1

Alternative ways of calling AWK (continued)

Example of calling program with no given files to scan:

```
awk ' $3 == 0 { print $1 }' <enter key>
```

AWK will apply the program to whatever you type next on the terminal, until you enter a <control-d> to terminate

An Example Session:

NOTE that the *bold lines* are what AWK returned and the *light lines* are what you entered

```
$ awk ' $3 == 0 { print $1 }'  
Beth    4.00    0  
Beth  
Dan     3.75    0  
Dan  
Flo     7.50   10  
Flo     7.50    0  
Flo  
...
```

Note that using the command like this is a good way to test the 'program' in a quick and dirty way.

Simple Output

The names of the columns are \$1, \$2, etc

The name of the ENTIRE line is \$0

If an action has no pattern, { print } is the same as { print \$0 }

As stated before, more than one field can be printed on the output line:

{ print \$1, \$3 } will print all entries in field 1 and field 3 of the file

By default, each **field *separated by a comma*** is ***separated by a space in the output***

By default, ***after a line is printed, a newline character is generated***

NF ... the Number of Fields

AWK has a built-in variable that contains the number of fields in the current input line...NOTE that lines *do not* have to contain the same number of fields

Given the following data in empfile:

Beth	4.00	0
Dan	3.00	0
Kathy	4.00	10
Mark	5.00	20
Mary	3.00	
Susie	6.00	10

```
$ awk { print NF, $1, $NF }, empfile
```

will produce:

3	Beth	0
3	Dan	0
3	Kathy	10
3	Mark	20
2	Mary	3.00
3	Susie	10

where **NF** is the *number of fields* in the current line
AND

\$NF represents the *contents of the last field* in the current line

Computing and Printing

Computations can be made inside the action statement:

Given the following data in empfile:

Beth	4.00	0
Dan	3.00	0
Kathy	4.00	10
Mark	5.00	20
Mary	3.00	10
Susie	6.00	10

```
$ awk { print $1, $2 * $3 }, empfile
```

will produce:

Beth	0
Dan	0
Kathy	40
Mark	100
Mary	30
Susie	60

Line Numbers

AWK provides a variable, **NR** that counts the number of lines *READ SO FAR*.

Given the following data in empfil:

Kathy	4.00	10
Mark	5.00	20
Mary	3.00	10
Susie	6.00	10

```
$ awk { NR, $0 }
```

will produce the following:

1	Kathy	4.00	10
2	Mark	5.00	20
3	Mary	3.00	10
4	Susie	6.00	10

Placing Text in the Output

Text can be embedded in an action

Given the following data in empfile:

Kathy	4.00	10
Mark	5.00	20
Mary	3.00	10
Susie	6.00	10

```
$ awk { print "Total pay for", $1, "is", $2 * $3 }, empfile
```

will produce the following:

```
Total pay for Kathy is 40
Total pay for Mark is 100
Total pay for Mary is 30
Total pay for Susie is 60
```

(NOTE the use of double-quotes in the program statement...not single quotes)

(NOTE that trailing zeros are not produced by the *print* statement)

Output using the *printf* statement

The *printf* statement has the form:

```
printf(format, value1, value2, ... , valuen)
```

where *format* is a string containing text to be printed *verbatim*, interspersed with *specifications* of how each of the *values* is to be printed.

A *specification* is % symbol followed by characters that control the format of a value.

Given the program:

```
{ printf("total pay for %s is $%.2f\n", $1, $2 * $3) }
```

%s will output a string (given the data in \$1)

%.2f will output a floating point number with two places to the right of the decimal point (given the result of \$2 * \$3)

\n will output a newline character

NOTE: The *print* statement will output a newline character automatically, and will also automatically place a space between field designators separated by commas at the output.

NOTE: *printf* does neither...YOU supply all WHITESPACE characters

Using printf (continued)

Given the following data in emp:

Kathy	4.00	10
Mark	5.00	20
Mary	3.00	10
Susie	6.00	10

the program:

```
$ awk { printf("Total pay for %s is $%.2f\n", $1, $2 * $3) }, emp
```

will produce:

```
Total pay for Kathy is $40.00
Total pay for Mark is $100.00
Total pay for Mary is $30.00
Total pay for Susie is $60.00
```

Likewise, the program:

```
$ awk { printf("%-8s $%.2f\n", $1, $2 * $3) }, emp will produce:
```

```
Kathy $ 40.00
Mark  $100.00
Mary  $ 30.00
Susie $ 60.00
```

where *name* is output *left-justified in a field 8 characters wide*, and *pay* is output as a number with *2 digits* after the *decimal point* in a *field 6 characters wide (the decimal point is counted as a character in the character count)*

more on printf later

Using AWK for Data Validation

Data validation is the act of making sure that the data in question contains reasonable values and is in the correct format.

Typically, data validation is conducted in the negative...that is, lines that *don't* meet given criteria are printed instead of those that do.

The following applies comparison patterns to apply plausibility tests on the data in empfile.dat:

```
NF != 3 { print $0, "number of fields not equal to 3" }
$2 < 3.35 { print $0, "rate below minimum" }
$2 > 10 { print $0, "rate exceeds $10 per hour" }
$3 < 0 { print $0, "negative hours worked" }
$3 > 60 { print $0, "too many hours worked" }
```

Note that if there are no errors in the file, there is no output

BEGIN and END

The special pattern *BEGIN* matches before first line of the first input file is read, and *END* matches after the last line of the last file has been processed.

Given the following:

```
BEGIN { print "NAME  RATE  HOURS"; print " " }
      { print }
```

the output will be:

NAME	RATE	HOURS
Kathy	4.00	10
Mark	5.00	20
Mary	3.00	10
Susie	6.00	10

NOTE: As shown above, *more than one statement* can be placed on a line as long as each statement (except the last) is *followed by a semi-colon* character.

NOTE: A blank line (new line) is produced by the statement:

print ""

User-defined Variables in AWK

In addition to built-in variables like NF, AWK allows users to create their own variables.

The following applies:

User-defined variables are not declared

User-defined variables used to store numbers are initialized to zero, if not given an initial value by the user

User-defined variables used to store strings are initialized to the *null string* (the string containing no characters)* if not given an initial value by the user.

*** NOTE: the *null string* (also known as the *empty string*) is a string containing **ZERO** characters...think of *two double-quote marks* with *nothing between them*...it's a valid string...it's just empty**

User-defined Variables in AWK (continued)

We'll use a variable called *emp* to count all employees in *empfile.dat* that worked more than 5 hours. Given the data:

Beth	4.00	0
Dan	3.00	5
Kathy	4.00	12
Mark	5.00	6
Mary	3.00	4
Susie	6.00	10

the following:

```
$3 > 5 { emp = emp + 1 }  
END { print emp, "employees worked over 5 hours" }
```

will yield:

3 employees worked over 5 hours

NR holds the *number of lines read so far*...its value after *all lines have been read* is the total number of lines read.

Here's another use of END:

```
END { print NR, "employees" }
```

will print the number of employees in the file

Some Other Ways to Use END

Given the data in emp.dat, we'll compute the average pay

Beth	4.25	4
Dan	3.75	5
Kathy	8.50	8
Mark	5.50	6
Mary	6.00	4
Susie	7.25	9

the following:

```
        { pay = pay + $2 * $3 }
END    { print NR, "employees"
        print "total pay for all employees is", pay
        print "average employee pay is", pay/NR
        }
```

The first action goes through the file and calculates the total pay, and stores it in the variable *pay*

After the last line is processed, the END action prints:

```
6 employees
total pay for all employees is 224
average employee pay is 37.33
```

obviously, printf would provide superior output

**CAN YOU SPOT THE POTENTIAL
SHOWSTOPPER IN THE CODE ABOVE?**

Some Other Ways to Use END (continued)

Given the data in emp.dat, we'll concatenate the names into a string separated by spaces

Beth	4.25	4
Dan	3.75	5
Kathy	8.50	8
Mark	5.50	6
Mary	6.00	4
Susie	7.25	9

the program:

```
        { names = names $1 " " }  
END { print names }
```

yields:

Beth Dan Kathy Mark Mary Susie

How does this thing work?

NOTE: the string contains a *space character* after Susie...why?

Yet another Way to Use END (continued)

Although NR retains its value in an END action, \$0 does not...so how can you print the last line? No sweat...do this:

```
{ last = $0 }  
END { print last }
```

Built-In Functions

There are all kinds...too many to enumerate here.

But you can look them up...

One tasty function is *length*

length returns the number of characters in a string

```
{ print $0, length($0) }
```

returns the number of chars in the current line...

CAREFUL: *length* doesn't count the newline char at the end of \$0...BUT newline char IS part of the line, so if you're counting chars, add 1 for the newline char in each line

Yet another Way to Use END (continued)

Count lines, words, and chars in a file, where the contents of a field is considered a word

Given the file emp.dat containing:

Beth	4.00	0
Dan	3.75	0
Kathy	4.00	10
Mark	5.00	20
Mary	5.50	22
Susie	4.25	18

```
    { nc = nc + length($0) + 1
      nw = nw + NF
    }
END { print, NR, "lines,", nw, "words," , nc, "chars" }
```

yields:

6 lines, 18 words, 77 chars

Control-Flow Statements

If-Else Statement

We'll compute the average pay for employees making more than \$6.00 per hour using an if-else statement

Given the file emp.dat containing:

Beth	4.00	0
Dan	3.75	0
Kathy	4.00	10
Mark	5.00	20
Mary	5.50	22
Susie	4.25	18

```
$2 > 6 { n = n + 1; pay = pay + $2 * $3 }
END { if (n > 0)
      print n, "employees, total pay is", pay,
          "average pay is", pay/n
    else
      print "nobody makes that much"
    }
```

What's the output?

Have we defended against division by zero?

NOTE: long lines can be continued over several lines by breaking it after a comma (*as in the first print action*)

Control-Flow Statements (continued)

While Loop

```
# this can be placed in a file called wloop
{   i = 1
    while ( i <= NF ) {
        print NF, $1 }
}
```

then called at the command line:

```
$ awk -f wloop, emp.dat
```

This will print the contents of each field in each record on a separate line

For Loop

```
# this can be placed in a file called floop
{ for (i = 1; i <=NF; i++) {
    print NF, $1 }
}
```

then called at the command line:

```
$ awk -f floop, emp.dat
```

This will print the contents of each field in each record on a separate line

Putting an AWK script in a file

Any AWK script commands can be placed in a file then called from the command line:

```
$ awk -f programName, input_file(s)
```

where **-f** tells awk to get instructions from the file **programName**

Printf Format Control Characters

Character	Print Expression As
c	ascii character
d	decimal integer
e	[-]d.dddddde[+-]dd
f	[-]ddd.ddddd
g	e or f conversion, whichever is shorter with non-significant zeros suppressed
s	string

Examples of Printf Specifications

<i>fmt</i>	<i>\$1</i>	<i>printf(fmt, \$1)</i>
<i>%c</i>	<i>97</i>	<i>a</i>
<i>%d</i>	<i>97.5</i>	<i>97</i>
<i>%5d</i>	<i>97.5</i>	<i>97</i>
<i>%e</i>	<i>97.5</i>	<i>9.750000e+01</i>
<i>%f</i>	<i>97.5</i>	<i>97.500000</i>
<i>%7.2f</i>	<i>97.5</i>	<i>97.50</i>
<i> %s </i>	<i>January</i>	<i> January </i>
<i> %10s </i>	<i>January</i>	<i> January </i>
<i> % -10s </i>	<i>January</i>	<i> January </i>
<i> .3s </i>	<i>January</i>	<i> Jan </i>
<i> %10.3s </i>	<i>January</i>	<i> Jan </i>
<i> % -10.3s </i>	<i>January</i>	<i> Jan </i>
<i> % % </i>	<i>January</i>	<i>%</i>

FS - Field Separator

The default field separator (with regard to input data) is the space

The value of FS is user definable, dependent on the context of the data being used.

For example, the FS used in the /etc/passwd file is :, so you would probably change FS to : when working with this file

Using getline to Read Input

getline can read input from standard in, a file, or a pipe

getline splits the fields of a file into an associative array

**Use this to read all the records in a file:
while (getline <"filename" > 0) ...**

where the > 0 is important because while it is true that there are lines to read, they will be read...save you from the problem of an endless loop if the file does not exist

Command-Line Variables

Command line arguments are stored in array called ARGV

The value of the built-in variable ARGC is the number of arguments+1

With the command line:

```
$ awk -f progfile a v=1 b
```

ARGC is 4, ARGV[0] contains awk, ARGV[1] contains a, ARGV[2] contains v=1, and ARGV[3] contains b

With the command line:

```
$ awk -F'\t' '$3 > 100' countries
```

ARGC is 2, ARGV[0] contains awk, ARGV[1] contains countries

NOTE: programName, -f, and any -F options are not counted as arguments

Command-Line Variables (continued)

This script echos command line arguments:

```
BEGIN {
    for (i = 1; i < ARGC; i++)
        printf "%s ", ARGV[i]
    printf "\n"
}
```

This script prints a sequence of integers:

```
# seq - print sequence if integers
#
#   input - arguments q, p q, or p q r
#           where q >= p and r > 0
#
#   output - integers 1 to q, p to q,
#            or p to q in steps of r
```

```
BEGIN {
    if (ARGC == 2)
        for (i = 1; i < ARGC; i++)
            print $i
    else if (ARGC == 3)
        for (i = ARGV[1]; i <= ARGV[2]; i++)
            print $i
    else if (ARGC == 4)
        for (i = ARGV[1]; i <= ARGV[2]; i+= ARGV[3])
            print $i
}
```

Command-Line Variables (continued)

the commands

```
awk -f seq 10
```

```
awk -f seq 1 10
```

```
awk -f seq 1 10 1
```

all generate integers 1 through 10