An Awk Primer/Awk Command-Line Examples

Introduction

It is easy to use Awk from the command line to perform simple operations on text files. Suppose I have a file named "coins.txt" that describes a coin collection. Each line in the file contains the following information:

- metal
- weight in ounces
- date minted
- country of origin
- description

The file has the following contents:

```
gold     1    1986  USA                American Eagle
gold     1    1908  Austria-Hungary    Franz Josef 100 Korona
silver  10    1981  USA                ingot
gold     1    1984  Switzerland        ingot
gold     1    1979  RSA                Krugerrand
gold     0.5  1981  RSA                Krugerrand
gold     0.1  1986  PRC                Panda
silver   1    1986  USA                Liberty dollar
gold     0.25 1986  USA                Liberty 5-dollar piece
silver   0.5  1986  USA                Liberty 50-cent piece
silver   1    1987  USA                Constitution dollar
gold     0.25 1987  USA                Constitution 5-dollar piece
gold     1    1988  Canada             Maple Leaf
```

I could then invoke Awk to list all the gold pieces as follows:

```
awk '/gold/' coins.txt
```

This tells Awk to search through the file for lines of text that contain the string "gold", and print them out. The result is:

```
gold     1    1986  USA                American Eagle
gold     1    1908  Austria-Hungary    Franz Josef 100 Korona
gold     1    1984  Switzerland        ingot
```

```
gold     1    1979  RSA               Krugerrand
gold     0.5  1981  RSA               Krugerrand
gold     0.1  1986  PRC               Panda
gold     0.25 1986  USA               Liberty 5-dollar piece
gold     0.25 1987  USA               Constitution 5-dollar piece
gold     1    1988  Canada            Maple Leaf
```

Printing the Descriptions

This is all very nice, a critic might say, but any "grep" or "find" utility can do the same thing. True, but Awk is capable of doing much more. For example, suppose I only want to print the description field, and leave all the other text out. I could then change my invocation of Awk to:

```
awk '/gold/ {print $5,$6,$7,$8}' coins.txt
```

This yields:

```
American Eagle
Franz Josef 100 Korona
ingot
Krugerrand
Krugerrand
Panda
Liberty 5-dollar piece
Constitution 5-dollar piece
Maple Leaf
```

Simplest Awk Program

This example demonstrates the simplest general form of an Awk program:

```
awk search pattern { program actions }
```

Awk searches through the input file line-by-line, looking for the search pattern. For each of these lines found, Awk then performs the specified actions. In this example, the action is specified as:

```
{print $5,$6,$7,$8}
```

The purpose of the `print` statement is obvious. The `$5`, `$6`, `$7`, and `$8` are **fields**, or "field variables", which store the words in each line of text by their numeric sequence. `$1`, for example, stores the first word in the line, `$2` has the second, and so on. By default a "word", or **record**, is defined as any string of printing characters separated by spaces.

Based on the structure of "coins.txt" (see above), the field variables are matched to each line of text in the file as follows:

```
metal:        $1
weight:       $2
date:         $3
country:      $4
description:  $5 through $8
```

The program action in this example prints the fields that contain the description. The description field in the file may actually include from one to four fields, but that's not a problem, since "print" simply ignores any undefined fields. The astute reader will notice that the "coins.txt" file is neatly organized so that the only piece of information that contains multiple fields is at the end of the line. This limit can be overcome by changing the field separator, explained later.

Awk's default program action is to print the entire line, which is what "print" does when invoked without parameters. This means that these three examples are the same:

```
awk '/gold/'
awk '/gold/ {print}'
awk '/gold/ {print $0}'
```

Note that Awk recognizes the field variable `$0` as representing the entire line. This is redundant, but it does have the virtue of making the action more obvious.

Conditionals

Now suppose I want to list all the coins that were minted before 1980. I invoke Awk as follows:

```
awk '{if ($3 < 1980) print $3, "    ",$5,$6,$7,$8}' coins.txt
```

This yields:

```
1908    Franz Josef 100 Korona
1979    Krugerrand
```

This new example adds a few new concepts:

Printing Lines

- If no search pattern is specified, Awk will match *all* lines in the input file, and perform the actions on each one.
- The `print` statement can display custom text (in this case, four spaces) simply by enclosing the text in quotes and adding it to the *parameter list*.
- An `if` statement is used to check for a certain condition, and the `print` statement is executed only if that condition is true.

There's a subtle issue involved here, however. In most computer languages, strings are strings, and numbers are numbers. There are operations that are unique to each, and one must be specifically converted to the other with conversion functions. You don't concatenate numbers, and you don't perform arithmetic operations on strings.

Awk, on the other hand, makes no strong distinction between strings and numbers. In computer-science terms, it isn't a "strongly-typed" language. All data in Awk are regarded as strings, but if that string also happens to represent a number, numeric operations can be performed on it. So we can perform an *arithmetic* comparison on the date field.

`BEGIN` and `END`

The next example prints out how many coins are in the collection:

```
awk 'END {print NR,"coins"}' coins.txt
```

This yields:

```
13 coins
```

The first new item in this example is the `END` statement. To explain this, I have to extend the general form of an Awk program.

Awk Programs

Every Awk program follows this format (each part being optional):

```
awk 'BEGIN { initializations } search pattern 1 { program actions } search pattern 2 { program actions }  ...  END { final actions
}' input file
```

The `BEGIN` clause performs any initializations required before Awk starts scanning the input file. The subsequent body of the Awk program consists of a series of search patterns, each with its own program action. Awk scans each line of the input file for each search pattern, and performs the appropriate actions for each string found. Once the file has been scanned, an `END` clause can be used to perform any final actions required.

So, this example doesn't perform any processing on the input lines themselves. All it does is scan through the file and perform a final action: print the number of lines in the file, which is given by the `NR` variable. `NR` stands for "number of records". `NR` is one of Awk's "pre-defined" variables. There are others, for example the variable `NF` gives the number of fields in a line, but a detailed explanation will have to wait for later.

Counting Money

Suppose the current price of gold is $425 per ounce, and I want to figure out the approximate total value of the gold pieces in the coin collection. I invoke Awk as follows:

```
awk '/gold/ {ounces += $2} END {print "value = $" 425*ounces}' coins.txt
```

This yields:

```
value = $2592.5
```

In this example, `ounces` is a variable I defined myself, or a "user-defined" variable. Almost any string of characters can be used as a variable name in Awk, as long as the name doesn't conflict with some string that has a specific meaning to Awk, such as `print` or `NR` or `END`. There is no need to declare the variable, or to initialize it. A variable handled as a string value is initialized to the "null string", meaning that if you try to print it, nothing will be there. A variable handled as a numeric value will be initialized to zero.

So the program action:

```
{ounces += $2}
```

sums the weight of the piece on each matched line into the variable `ounces`. Those who program in C should be familiar with the `+=` operator. Those who don't can be assured that this is just a shorthand way of saying:

```
{ounces = ounces + $2}
```

The final action is to compute and print the value of the gold:

```
END {print "value = $" 425*ounces}
```

The only thing here of interest is that the two print parameters—the literal `value = $` and the expression `425*ounces`—are separated by a space, not a comma. This concatenates the two parameters together on output, without any intervening spaces.

Practice

1. Try modifying one of the above programs to calculate and display the total amount (in ounces) of gold and silver, separately but with one program. You will have to use two pairs of pattern/action.
2. Write an Awk program that finds the average weight of all coins minted in the USA.
3. Write an Awk program that reprints its input with line numbers before each line of text.

In the next chapter, we learn how to write Awk programs that are longer than one line.

If you do not give Awk an input file, it will allow you to type input directly to your program. Pressing CTRL-D will quit.