Common threads: Awk by example, Part 3: String functions and ... checkbooks?

Daniel Robbins

April 01, 2001

In this conclusion to the awk series, Daniel introduces you to awk's important string functions, and then shows you how to write a complete checkbook-balancing program from scratch. Along the way, you'll learn how to write your own functions and use awk's multidimensional arrays. By the end of this article, you'll have even more awk experience, allowing you to create more powerful scripts.

View more content in this series

Formatting output

While awk's print statement does do the job most of the time, sometimes more is needed. For those times, awk offers two good old friends called printf() and sprintf(). Yes, these functions, like so many other awk parts, are identical to their C counterparts. printf() will print a formatted string to stdout, while sprintf() returns a formatted string that can be assigned to a variable. If you're not familiar with printf() and sprintf(), an introductory C text will quickly get you up to speed on these two essential printing functions. You can view the printf() man page by typing "man 3 printf" on your Linux system.

Here's some sample awk sprintf() and printf() code. As you can see, everything looks almost identical to C.

```
x=1 b="foo" printf("%s got a %d on the last test\n","Jim",83) myout=("%s-%d",b,x) print myout
```

This code will print:

```
Jim got a 83 on the last test foo-1
```

String functions

Awk has a plethora of string functions, and that's a good thing. In awk, you really need string functions, since you can't treat a string as an array of characters as you can in other languages like C, C++, and Python. For example, if you execute the following code:

```
mystring="How are you doing today?" print mystring[3]
```

You'll receive an error that looks something like this:

```
awk: string.gawk:59: fatal: attempt to use scalar as array
```

Oh, well. While not as convenient as Python's sequence types, awk's string functions get the job done. Let's take a look at them.

First, we have the basic length() function, which returns the length of a string. Here's how to use it:

```
print length(mystring)
```

This code will print the value:

```
24
```

OK, let's keep going. The next string function is called index, and will return the position of the occurrence of a substring in another string, or it will return 0 if the string isn't found. Using mystring, we can call it this way:

```
print index(mystring,"you")
```

Awk prints:

```
9
```

We move on to two more easy functions, tolower() and toupper(). As you might guess, these functions will return the string with all characters converted to lowercase or uppercase respectively. Notice that tolower() and toupper() return the new string, and don't modify the original. This code:

```
print tolower(mystring) print toupper(mystring) print mystring
```

....will produce this output:

```
how are you doing today? HOW ARE YOU DOING TODAY? How are you doing today?
```

So far so good, but how exactly do we select a substring or even a single character from a string? That's where substr() comes in. Here's how to call substr():

```
mysub=substr(mystring,startpos,maxlen)
```

mystring should be either a string variable or a literal string from which you'd like to extract a substring. startpos should be set to the starting character position, and maxlen should contain the maximum length of the string you'd like to extract. Notice that I said *maximum length*; if length(mystring) is shorter than startpos+maxlen, your result will be truncated. substr() won't modify the original string, but returns the substring instead. Here's an example:

```
print substr(mystring,9,3)
```

Awk will print:

```
you
```

If you regularly program in a language that uses array indices to access parts of a string (and who doesn't), make a mental note that substr() is your awk substitute. You'll need to use it to extract single characters and substrings; because awk is a string-based language, you'll be using it often.

Now, we move on to some meatier functions, the first of which is called match(). match() is a lot like index(), except instead of searching for a substring like index() does, it searches for a regular expression. The match() function will return the starting position of the match, or zero if no match is found. In addition, match() will set two variables called RSTART and RLENGTH. RSTART contains the return value (the location of the first match), and RLENGTH specifies its span in characters (or -1 if no match was found). Using RSTART, RLENGTH, substr(), and a small loop, you can easily iterate through every match in your string. Here's an example match() call:

```
print match(mystring,/you/), RSTART, RLENGTH
```

Awk will print:

```
9 9 3
```

String substitution

Now, we're going to look at a couple of string substitution functions, sub() and gsub(). These guys differ slightly from the functions we've looked at so far in that they *actually modify the original string*. Here's a template that shows how to call sub():

```
sub(regexp,replstring,mystring)
```

When you call sub(), it'll find the first sequence of characters in mystring that matches regexp, and it'll replace that sequence with replstring. sub() and gsub() have identical arguments; the only way they differ is that sub() will replace the first regexp match (if any), and gsub() will perform a global replace, swapping out all matches in the string. Here's an example sub() and gsub() call:

```
sub(/o/,"O",mystring) print mystring mystring="How are you doing today?" gsub(/o/,"O",mystring) print
mystring
```

We had to reset mystring to its original value because the first sub() call modified mystring directly. When executed, this code will cause awk to output:

```
HOw are you doing today? HOw are yOu dOing tOday?
```

Of course, more complex regular expressions are possible. I'll leave it up to you to test out some complicated regexps.

We wrap up our string function coverage by introducing you to a function called split(). split()'s job is to "chop up" a string and place the various parts into an integer-indexed array. Here's an example split() call:

```
numelements=split("Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec",mymonths,",")
```

When calling split(), the first argument contains the literal string or string variable to be chopped. In the second argument, you should specify the name of the array that split() will stuff the chopped parts into. In the third element, specify the separator that will be used to chop the strings up. When split() returns, it'll return the number of string elements that were split. split() assigns each one to an array index starting with one, so the following code:

```
print mymonths[1],mymonths[numelements]
```

....will print:

```
Jan Dec
```

Special string forms

A quick note -- when calling length(), sub(), or gsub(), you can drop the last argument and awk will apply the function call to $0 (the entire current line). To print the length of each line in a file, use this awk script:

```
{ print length() }
```

Financial fun

A few weeks ago, I decided to write my own checkbook balancing program in awk. I decided that I'd like to have a simple tab-delimited text file into which I can enter my most recent deposits and withdrawals. The idea was to hand this data to an awk script that would automatically add up all the amounts and tell me my balance. Here's how I decided to record all my transactions into my "ASCII checkbook":

```
23 Aug 2000 food - - Y Jimmy's Buffet 30.25
```

Every field in this file is separated by one or more tabs. After the date (field 1, $1), there are two fields called "expense category" and "income category". When I'm entering an expense like on the above line, I put a four-letter nickname in the exp field, and a "-" (blank entry) in the inc field. This signifies that this particular item is a "food expense" :) Here's what a deposit looks like:

```
23 Aug 2000 - inco - Y Boss Man 2001.00
```

In this case, I put a "-" (blank) in the exp category, and put "inco" in the inc category. "inco" is my nickname for generic (paycheck-style) income. Using category nicknames allows me to generate a breakdown of my income and expenditures by category. As far as the rest of the records, all the other fields are fairly self-explanatory. The cleared? field ("Y" or "N") records whether the

transaction has been posted to my account; beyond that, there's a transaction description, and a positive dollar amount.

The algorithm used to compute the current balance isn't too hard. Awk simply needs to read in each line, one by one. If an expense category is listed but there is no income category (it's "-"), then this item is a debit. If an income category is listed, but no expense category ("-") is there, then the dollar amount is a credit. And, if there is both an expense and income category listed, then this amount is a "category transfer"; that is, the dollar amount will be subtracted from the expense category and added to the income category. Again, all these categories are virtual, but are very useful for tracking income and expenditures, as well as for budgeting.

The code

Time to look at the code. We'll start off with the first line, the BEGIN block and a function definition:

balance, part 1

```
#!/usr/bin/env awk -f BEGIN { FS="\t+" months="Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec" } function
monthdigit(mymonth) { return (index(months,mymonth)+3)/4 }
```

Adding the first "#!..." line to any awk script will allow it to be directly executed from the shell, provided that you "chmod +x myscript" first. The remaining lines define our BEGIN block, which gets executed before awk starts processing our checkbook file. We set FS (the field separator) to "\t+", which tells awk that the fields will be separated by one or more tabs. In addition, we define a string called months that's used by our monthdigit() function, which appears next.

The last three lines show you how to define your own awk function. The format is simple -- type "function", then the function name, and then the parameters separated by commas, inside parentheses. After this, a "{ }" code block contains the code that you'd like this function to execute. All functions can access global variables (like our months variable). In addition, awk provides a "return" statement that allows the function to return a value, and operates similarly to the "return" found in C, Python, and other languages. This particular function converts a month name in a 3-letter string format into its numeric equivalent. For example, this:

```
print monthdigit("Mar")
```

....will print this:

```
3
```

Now, let's move on to some more functions.

Financial functions

Here are three more functions that perform the bookkeeping for us. Our main code block, which we'll see soon, will process each line of the checkbook file sequentially, calling one of these functions so that the appropriate transactions are recorded in an awk array. There are three basic

kinds of transactions, credit (doincome), debit (doexpense) and transfer (dotransfer). You'll notice that all three functions accept one argument, called mybalance. mybalance is a placeholder for a two-dimensional array, which we'll pass in as an argument. Up until now, we haven't dealt with two-dimensional arrays; however, as you can see below, the syntax is quite simple. Just separate each dimension with a comma, and you're in business.

We'll record information into "mybalance" as follows. The first dimension of the array ranges from 0 to 12, and specifies the month, or zero for the entire year. Our second dimension is a four-letter category, like "food" or "inco"; this is the actual category we're dealing with. So, to find the entire year's balance for the food category, you'd look in mybalance[0,"food"]. To find June's income, you'd look in mybalance[6,"inco"].

balance, part 2

```
function doincome(mybalance) { mybalance[curmonth,$3] += amount mybalance[0,$3] += amount }
function doexpense(mybalance) { mybalance[curmonth,$2] -= amount mybalance[0,$2] -= amount } function
dotransfer(mybalance) { mybalance[0,$2] -= amount mybalance[curmonth,$2] -= amount mybalance[0,$3] += amount
mybalance[curmonth,$3] += amount }
```

When doincome() or any of the other functions are called, we record the transaction in two places -- mybalance[0,category] and mybalance[curmonth, category], the entire year's category balance and the current month's category balance, respectively. This allows us to easily generate either an annual or monthly breakdown of income/expenditures later on.

If you look at these functions, you'll notice that the array referenced by mybalance is passed in my reference. In addition, we also refer to several global variables: curmonth, which holds the numeric value of the month of the current record, $2 (the expense category), $3 (the income category), and amount ($7, the dollar amount). When doincome() and friends are called, all these variables have already been set correctly for the current record (line) being processed.

The main block

Here's the main code block that contains the code that parses each line of input data. Remember, because we have set FS correctly, we can refer to the first field as $1, the second field as $2, etc. When doincome() and friends are called, the functions can access the current values of curmonth, $2, $3 and amount from inside the function. Take a look at the code and meet me on the other side for an explanation.

balance, part 3

```
{ curmonth=monthdigit(substr($1,4,3)) amount=$7 #record all the categories encountered if ( $2 != "-" )
globcat[$2]="yes" if ( $3 != "-" ) globcat[$3]="yes" #tally up the transaction properly if ( $2 == "-" )
{ if ( $3 == "-" ) { print "Error: inc and exp fields are both blank!" exit 1 } else { #this is income
doincome(balance) if ( $5 == "Y" ) doincome(balance2) } } else if ( $3 == "-" ) { #this is an expense
doexpense(balance) if ( $5 == "Y" ) doexpense(balance2) } else { #this is a transfer dotransfer(balance) if
( $5 == "Y" ) dotransfer(balance2) } }
```

In the main block, the first two lines set curmonth to an integer between 1 and 12, and set amount to field 7 (to make the code easier to understand). Then, we have four interesting lines, where we

write values into an array called globcat. globcat, or the global categories array, is used to record all those categories encountered in the file -- "inco", "misc", "food", "util", etc. For example, if $2 == "inco", we set globcat["inco"] to "yes". Later on, we can iterate through our list of categories with a simple "for (x in globcat)" loop.

On the next twenty or so lines, we analyze fields $2 and $3, and record the transaction appropriately. If $2=="-" and $3!="-", we have some income, so we call doincome(). If the situation is reversed, we call doexpense(); and if both $2 and $3 contain categories, we call dotransfer(). Each time, we pass the "balance" array to these functions so that the appropriate data is recorded there.

You'll also notice several lines that say "if ($5 == "Y"), record that same transaction in *balance2*". What exactly are we doing here? You'll recall that $5 contains either a "Y" or a "N", and records whether the transaction has been posted to the account. Because we record the transaction to balance2 only if the transaction has been posted, balance2 will contain the actual account balance, while "balance" will contain all transactions, whether they have been posted or not. You can use balance2 to verify your data entry (since it should match with your current account balance according to your bank), and use "balance" to make sure that you don't overdraw your account (since it will take into account any checks you have written that have not yet been cashed).

Generating the report

After the main block repeatedly processes each input record, we now have a fairly comprehensive record of debits and credits broken down by category and by month. Now, all we need to do is define an END block that will generate a report, in this case a modest one:

```
END { bal=0 bal2=0 for (x in globcat) { bal=bal+balance[0,x] bal2=bal2+balance2[0,x] } printf("Your
available funds: %10.2f\n", bal) printf("Your account balance: %10.2f\n", bal2) }
```

This report prints out a summary that looks something like this:

```
Your available funds: 1174.22 Your account balance: 2399.33
```

In our END block, we used the "for (x in globcat)" construct to iterate through every category, tallying up a master balance based on all the transactions recorded. We actually tally up two balances, one for available funds, and another for the account balance. To execute the program and process your own financial goodies that you've entered into a file called "mycheckbook.txt", put all the above code into a text file called "balance", "chmod +x balance", and then type "./ balance mycheckbook.txt". The balance script will then add up all your transactions and print out a two-line balance summary for you.

Upgrades

I use a more advanced version of this program to manage my personal and business finances. My version (which I couldn't include here due to space limitations) prints out a monthly breakdown of income and expenses, including annual totals, net income and a bunch of other stuff. Even better, it outputs the data in HTML format, so that I can view it in a Web browser :) If you find this program

useful, I encourage you to add these features to this script. You won't need to configure it to *record* any additional information; all the information you need is already in balance and balance2. Just upgrade the END block, and you're in business!

I hope you've enjoyed this series. For more information on awk, check out the resources listed below.

Related topics

- Read Daniel's earlier installments in the awk series: Awk by example, Part 1 and Part 2 on *developerWorks*.
- 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.