Gnu Awk - Part 10 (HPR Show 2526)

Dave Morriss

Contents

Introduction

This is the tenth episode of the "Learning Awk" series which is being produced by b-yeezi and myself.

In this episode I want to talk more about the use of arrays in GNU Awk and then I want to examine some real-world examples of the use of `awk`.

A bit more about arrays

A recap

We know from earlier in the series that arrays in `awk` are *associative*. That is, the index used to refer to an element is a string. The contents of each array element may be a number or a string (or nothing). An associative array is also called a *hash*. An array *index* is also referred to as a *subscript*.

We also know that array elements are referred to with expressions such as:

array[*index*]

so, `fruit["apple"]` means the element of the array `fruit` which is indexed by the string `"apple"`. The index value is actually an expression, so it can be arbitrarily complex, such as:

```
ind1 = "app"
ind2 = "le"
print fruit[ind1 ind2]
```

Here the two strings `"app"` and `"le"` are concatenated to make the index `"apple"`.

We saw earlier in the series that the presence of an array element is checked with an expression using:

index `in` *array*

So an example might be:

```
if ("apple" in fruit)
    print fruit["apple"]
```

Looping through the elements of an array is achieved with the specialised `for` statement as we saw in an earlier episode:

```
for (ind in fruit)
    print fruit[ind]
```

Using numbers as array subscripts

In `awk` array subscripts are always strings. If a number is used then this is converted into a string. This is not a problem with statements like the following:

```
data[42] = 8388607
```

The integer number `42` is converted into the string `"42"` and everything works as normal.

However, `awk` can handle other number bases. For example, in common with many other programming languages, a leading zero denotes an octal number, making `data[052]` the same as `data[42]` (because decimal 42 is octal 52).

Also `data[0x2A]` is the same as `data[42]` because hexadecimal `2A` is decimal 42.

The way in which numbers are converted into strings in `awk` is important to understand. A built-in variable called `CONVFMT` defines the conversion for floating point numbers. Behind the scenes the function `sprintf` is used. (This is like `printf` which we saw in episode 9, but it returns a formatted string rather than printing anything.)

The default value for `CONVFMT` is `"%.6g"` which means (according to the manual) to *print a number in either scientific notation or in floating-point notation, whichever uses fewer characters.* The number 6 aims to format the number in a width of 6 characters (plus the decimal point). The setting of `CONVFMT` can be adjusted in the script if desired.

Knowing this the index can be determined in cases like this:

```
$ awk 'BEGIN{ x=100/3; data[x]="custard"; print x, data[x] }'
33.3333 custard
```

However, things get a little weird in this case:

```
$ awk 'BEGIN{ x=1000000000/3; data[x]="prunes"; print x, data[x] }'
3.33333e+08 prunes
```

The thing to be careful of is adjusting `CONVFMT` between storing and retrieving an array element!

What if the subscript is uninitialised?

The GNU Awk User's Guide mentions this. An uninitialised variable treated as a number is zero, but treated as a string is a null string `""`. The following script is in the file awk10_ex1.awk which may be downloaded:

```
#!/usr/bin/awk -f
{
    a[l] = $0
    l++
    print NR" "$0
}
END{
    print "Numeric subscripts:"
    for (i = l - 1; i >= 0; i--)
        print i": "a[i]

    print "Actual subscripts:"
    for (i in a)
        print i": "a[i]
}
```

This can lead to unexpected results:

```
$ echo -e "A\nB\nC" | ./awk10_ex1.awk
1 A
2 B
3 C
Numeric subscripts:
2: C
```

```
1: B
0:
Actual subscripts:
: A
0:
1: B
2: C
```

The variable `l` is used as the index to the array `a`. It is uninitialised the first time it is used so the string it provides is an empty string, which is a valid array index. Then it is incremented and it then takes numeric values. The main rule prints each line as it receives it just to prove it's actually seeing all three lines.

In the `END` rule the array is printed (in reverse order) using numeric indexes 2, 1 and zero. There is nothing in element zero.

Then the array is printed again using the "*index in array*" method. Notice how the letter `A` is there with an empty index. Notice also that there is an element with index zero too. That was created in the previous loop since accessing a non-existent array element creates it!

Had the two lines in the main rule been replaced as shown the outcome would have been more predictable:

```
a[l] = $0
l++
```

Replacement:

```
a[l++] = $0
```

Remembering that `l++` returns the value of `l` then increments it, this forces the first value returned to be zero because it is a numeric expression.

Deleting array elements

There is a `delete` statement which can delete a given array element. For example, in the above demonstration of subscript issues, the spurious element could have been deleted with:

```
delete a[0]
```

The generic format is:

`delete` *array*[*index*]

We already saw that array elements with empty subscripts or empty values can exist in an array, so we know that making an element empty does not delete it.

An entire array can be deleted with the generic statement:

`delete` *array*

The array remains declared but is empty, so re-using its name as an ordinary (scalar) variable after using `delete` on it will result in an error.

Splitting strings into arrays

There are two functions in `awk` which generate arrays from strings by splitting them up by some criterion. The functions are: `split` and `patsplit`. We will look at `split` in this episode and `patsplit` in a subsequent one.

`split`

The general format of the `split` function is:

`split`(*string*, *array* [, *fieldsep* [, *seps*]])

The first two arguments are mandatory but the second two are optional.

The function divides *string* into pieces separated by *fieldsep* and stores the pieces in *array* and the separator strings in the *seps* array (a GNU Awk extension).

Successive pieces are placed in *array*`[1]`, *array*`[2]`, and so on. The *array* is emptied before the splitting begins.

If *fieldsep* is omitted then the value of the built-in variable `FS` is used, so `split` can be seen as a method of generating fields from a string in a similar way to the main field processing that `awk` performs. If *fieldsep* is provided than it is a regular expression (again in the same way as `FS`).

The *seps* array is used to hold each of the separators. If *fieldsep* is a single space then any leading white space goes into *seps*`[0]` and any trailing white space goes into *seps*`[n]`, where `n` is the number of number of elements in *array*.

The function *split* returns the number of pieces placed in *array*.

Example of using `split`

The following script is in the file awk10_ex2.awk which may be downloaded:

```
#!/usr/bin/awk -f
{
    lines[NR] = $0
}

END{
    for (i in lines) {
        split(lines[i],flds,/ *, */,seps)
        for (j in flds)
            printf "|%s| (%s)\n",flds[j],seps[j]
    }
```

```
}
```

It reads lines into an array called `lines` using the record number as the index. In the `END` rule it processes this array, splitting each line into another array called `flds` and the separators into an array called `seps`.

The *fieldsep* value is a regular expression consisting of a comma surrounded by any number of spaces. The `flds` array is printed in delimiters to demonstrate that any leading and trailing spaces have been removed. The `seps` array is appended to each output line enclosed in parentheses so you can see what was captured there.

Here is what happens when the script is run:

```
$ echo -e "A,B,C\nD,   E   ,F" | ./awk10_ex2.awk
|A| (,)
|B| (,)
|C| ()
|D| (,   )
|E| (   ,)
|F| ()
```

Real-world Examples

The following example scripts are not specifically about the use of arrays in `awk`. This is more of an attempt to demonstrate some real-world `awk` scripts for reference.

Scanning a log file

I have a script I wrote to add tags and summaries to HPR episodes that have none. I seem to mention this project every month on the Community News show! The script receives email messages with updates, and keeps a log with lines that look like this as it processes them:

```
2018/02/19 04:17:21 [INFO] Moving /home/dave/MailSpool/episode-736.eml to 'processed'
2018/02/19 04:17:21 [INFO] 736:tags:summary
```

Note: if you are wondering about the times they are local to the server, based in California USA, on which the script is run. I run things from the UK timezone (UTC or UTC+1).

I like to add a report on the number of tags and summaries processed each month to the Community News show notes, so I wanted to scan this log file for the month's total.

Originally I used a pipeline with `grep` and `wc` but the task is well suited to `awk`. This was my solution (with added line numbers for reference):

```
 1: awk '
 2: BEGIN{
 3:     re = "^" strftime("%Y/%m/") ".. .* [0-9]{1,4}:"
 4:     count = 0
 5: }
 6: $0 ~ re {
 7:     printf "%02d %s\n",++count,$0
 8: }
 9: END{
10:     print "Additions",count
11: }
12: ' process_mail_tags.log
```

- In the `BEGIN` (lines 2-5) rule a regular expression is defined in the variable `re`.
  - This starts with a '`^`' character which anchors the expression to the start of the line.
  - This is followed by part of the date generated with the built-in function `strftime`. Here we generate the current year and the current month number and a slash.
  - Two dots follow which cater for the day number, then there is a space and '`.*`' meaning zero or more characters.
  - This is followed by a space then between one and four digits. This matches the show number after the '`[INFO]`' part.
  - The expression ends with a colon which matches the one after the show number
- In the rule a variable `count` is initialised to zero (not strictly necessary but good programming practice)
- The main rule for processing the input file (lines 6-8) matches each line against the regular expression. If it matches the line is printed preceded by the current value of `count` (which is pre-incremented before being printed).
- The `END` rule (lines 9-11) prints the final value of `count`.

Running this towards the end of February 2018 we get:

```
01: 2018/02/05 01:19:09 [INFO] 788:summary:tags
02: 2018/02/05 05:17:27 [INFO] 1683:tags
03: 2018/02/05 06:15:16 [INFO] 1663:tags
04: 2018/02/05 06:15:16 [INFO] 1666:tags
05: 2018/02/05 06:15:16 [INFO] 1668:tags
06: 2018/02/05 06:15:16 [INFO] 1669:tags
07: 2018/02/05 06:22:43 [INFO] 1693:tags
08: 2018/02/05 06:52:13 [INFO] 1550:tags
09: 2018/02/05 06:52:13 [INFO] 1551:tags
10: 2018/02/05 06:52:13 [INFO] 1552:tags
```

```
11: 2018/02/05 06:52:13 [INFO] 1554:tags
12: 2018/02/05 06:52:13 [INFO] 1556:tags
13: 2018/02/05 06:52:13 [INFO] 1559:tags
14: 2018/02/05 14:33:46 [INFO] 1540:tags
15: 2018/02/05 14:33:46 [INFO] 1541:tags
16: 2018/02/05 14:33:46 [INFO] 1543:tags
17: 2018/02/05 14:33:46 [INFO] 1547:tags
18: 2018/02/05 14:33:46 [INFO] 1549:tags
19: 2018/02/17 11:44:56 [INFO] 798:tags:summary
20: 2018/02/18 02:55:53 [INFO] 0021:summary:tags
21: 2018/02/19 04:17:21 [INFO] 736:tags:summary
22: 2018/02/25 03:32:45 [INFO] 1480:tags
23: 2018/02/25 03:32:45 [INFO] 1489:summary:tags
Additions 23
```

Of course, I would not run this `awk` script on the command line as shown here. I'd place it in a Bash script to simplify the typing, but I will not demonstrate that here.

Parsing a tab-delimited file

I am currently looking after the process of uploading HPR episodes to the Internet Archive (IA) - `archive.org`. To manage this I use a Python library called *internetarchive* and a command line tool called *ia*. The `ia` tool lets me interrogate the archive, returning data about shows that have been uploaded as well as allowing me to upload and change them.

In some cases I find it necessary to replace the audio formats which have been generated automatically by `archive.org` with copies generated by the HPR software. This is because we want to ensure these audio files contain metadata (*audio tags*). The shows generated by `archive.org` are converted from the WAV file we upload in a process referred to as *derivation*, and contain no metadata.

I needed to be able to tell which HPR episodes had derived audio and which had original audio. The `ia` tool could do this but in a format which was difficult to parse, so I wrote an `awk` script to do it for me.

The data I needed to parse consists of tab-delimited lines. The first line contains the names of all of the columns. However, some the columns were not always present or were in different orders, so this required a little more work to parse.

Here is a sample of the input file format:

```
$ ia list -va hpr2450 | head -3
name    sha1    format  btih    height  source  length  width   mtime   crc32   size    bitr
hpr2450.afpk    b71f63ef1e8c359b3f0f7a546835919a8a7889da        Columbia Peaks
hpr2450.flac    cd917c46eaf22f0ec0253bd018b475380e83ce7e        Flac            0       deri
```

The following script, called `parse_ia_audio.awk`, was what I produced to parse this data.

```
#!/usr/bin/awk -f

#-------------------------------------------------------------------------------
# Process tab-delimited data from the Internet Archive with a field name
# header, reporting particular fields. The algorithm is general though this
# instance is specific.
#
# In this case we extract only the audio files
#
# This script is meant to be used thus:
#       $ ia list -va hpr2450 | ./parse_ia_audio.awk
#       hpr2450.flac derivative
#       hpr2450.mp3  derivative
#       hpr2450.ogg  derivative
#       hpr2450.opus original
#       hpr2450.spx  original
#       hpr2450.wav  original
#
#-------------------------------------------------------------------------------

BEGIN {
    FS = "\t"
}

#
# Read the header line and collect the fields into an array such that a search
# by field name returns the field number.
#
NR == 1 {
    for (i = 1; i <= NF; i++) {
        fld[$i] = i
    }
}

#
# Read the rest of the data, reporting only the lines relating to audio files
# and print the fields 'name' and 'source'
#
NR > 1 && $(fld["name"]) ~ /[^.]\.(flac|mp3|ogg|opus|spx|wav)/ {
    printf "%-15s %s\n",$(fld["name"]),$(fld["source"])
}
```

The `BEGIN` rule defines the field delimiter as the TAB character.

The first rule runs only when the first record is encountered. This is the header with the names of the columns (fields). A `for` loop scans the fields which have been split up by `awk`'s usual record splitting. The fields are named `$1`, `$2` etc. The variable `i` increments from 1 to however many fields there are in the record and stores the field numbers in the array `fld` indexed by the contents of the field.

The end result will be:

```
fld["name"] = 1
fld["sha1"] = 2
fld["format"] = 3
etc
```

The second rule is invoked if two conditions are met:

- The record number is greater than 1
- The field numbered whatever the header `"name"` returned (1 in the example) ends with one of `flac`, `mp3`, `ogg`, `opus`, `spx`, `wav`

This rule prints the fields indexed by the column names `"name"` and `"source"`. The first comment in the script shows what this will look like.

Note the use of expressions like:

```
$(fld["name"])
```

Here `awk` will find the value stored in `fld["name"]` (1 in the example data) and will reference the field called `$(1)`, which is another way of writing `$1`. The parentheses are necessary to remove ambiguity.

So, the script is just printing columns for certain selected lines, but is able to cope with the columns being in different positions at different times because it prints them "by name".

Most of the queries handled by the Internet Archive API return JSON-format results (not something that `awk` can easily parse), but for some reason this one returns a varying tab-delimited file. Still, `awk` was able to come to the rescue!

Links

- *GNU Awk User's Guide*
- Previous shows in this series on HPR:
  - "*Gnu Awk - Part 1*" - episode 2114
  - "*Gnu Awk - Part 2*" - episode 2129
  - "*Gnu Awk - Part 3*" - episode 2143
  - "*Gnu Awk - Part 4*" - episode 2163
  - "*Gnu Awk - Part 5*" - episode 2184
  - "*Gnu Awk - Part 6*" - episode 2238
  - "*Gnu Awk - Part 7*" - episode 2330

  - "*Gnu Awk - Part 8*" - episode 2438
  - "*Gnu Awk - Part 9*" - episode 2476
- Resources:
  - ePub version of these notes
  - PDF version of these notes
  - awk10_ex1.awk
  - awk10_ex2.awk