Gnu Awk - Part 6 (HPR Show 2238)

Dave Morriss

Contents

Introduction . . . 1
Recap of the last episode . . . 2
- Regular expressions . . . 2
- Replacement . . . 2

More about regular expressions . . . 3
- More regular expression operators . . . 3

Functions . . . 4
- The `sub` function . . . 4
  - Examples using `sub` . . . 5
- The `gsub` function . . . 5
  - Examples using `gsub` . . . 5
- The `gensub` function . . . 6
  - First argument: *regexp* . . . 6
  - Second argument: *replacement* . . . 6
  - Third argument: *how* . . . 7
  - Fourth argument: *target* . . . 7
  - Examples using `gensub` . . . 7

Example script . . . 8
Warning for `sed` users . . . 10
Summary . . . 10
Links . . . 10

Introduction

This is the sixth episode of the "Learning Awk" series that b-yeezi and I are doing.

Recap of the last episode

Regular expressions

In the last episode we saw regular expressions in the '*pattern*' part of a '*pattern {action}*' sequence. Such a sequence is called a '*RULE*', (as we have seen in earlier episodes).

```
$1 ~ /p[elu]/ {print $0}
```

Meaning: *If field 1 contains a 'p' followed by one of 'e', 'l' or 'u' print the whole line.*

```
$2 ~ /e{2}/ {print $0}
```

Meaning: *If field 2 contains two instances of letter 'e' in sequence, print the whole line.*

It is usual to enclose the regular expression in slashes, which make it a *regexp constant* (see the GNU Manual for the details of these constants).

We had a look at many of the *operators* used in regular expressions in episode 5. Unfortunately, some small errors crept into the list of operators mentioned in that episode. These are incorrect:

- ~~\A~~ (beginning of a string)
- ~~\z~~ (end of a string)
- ~~\b~~ (on a word boundary)
- ~~\d~~ (any digit)

The first two operators exist, as does the last one, but only in languages like Perl and Ruby, but not in GNU Awk.

For the '\b' sequence the GNU manual says:

> *In other GNU software, the word-boundary operator is '\b'. However, that conflicts with the awk language's definition of '\b' as backspace, so gawk uses a different letter. An alternative method would have been to require two backslashes in the GNU operators, but this was deemed too confusing. The current method of using '\y' for the GNU '\b' appears to be the lesser of two evils.*

The corrected list of operators is discussed later in this episode.

Replacement

Last episode we saw the built-in functions that use regular expressions for manipulating strings. These are `sub`, `gsub` and `gensub`. Regular expressions are used in other functions but we will look at them later.

We will be looking at `sub`, `gsub` and `gensub` in more detail in this episode.

More about regular expressions

More regular expression operators

We have seen that the regular expressions in GNU Awk use certain characters to denote concepts. For example, '.' is not a full-stop (period) in a regular expression, but means *any character*. This special meaning can be turned off by preceding the character by a backslash '\'. Since a backslash is itself a *special* character, if you need an actual backslash in a regular expression then precede it with a backslash ('\\'). We will demonstrate how the backslash might be used in the examples later.

Note that (as with GNU sed) some regular expression operators consist of a backslash followed by a character.

The following table summarises some of the regular expression operators, including some we have already encountered.

Expression	Meaning
any character	A single ordinary character matches itself
.	Matches any character
*****	Matches a sequence of zero or more instances of the preceding item
[*list*]	Matches any single character in *list*: for example, [aeiou] matches all vowels
[^*list*]	A leading '^' reverses the meaning of *list*, so that it matches any single character not in
^	Matches the beginning of the line (anchors the search at the start)
$	Matches the end of the line (anchors the search at the end)
+	Similar to ***** but matches a sequence of one or more instances of the preceding item
?	Similar to ***** but matches a sequence of zero or one instance of the preceding item
{i}	Matches exactly `i` sequences (`i` is a decimal integer)
{i,j}	Matches between `i` and `j` sequences, inclusive
{i,}	Matches `i` or more sequences, inclusive
(regexp)	Groups the inner *regexp*. Allows it to be followed by a postfix operator, or can be used f
regexp1|regexp2	Matches *regexp1* or *regexp2*, **|** is used to separate alternatives

The expressions '[*list*]' and '[^*list*]' are known as *bracket expressions* in GNU Awk. They represent a single character chosen from the *list*.

To include the characters '\', ']', '-' or '^' in the list precede them with a backslash.

The *character classes* like '[:alnum:]' were dealt with in episode 5. These can only be used in *bracket expressions* and represent a single character. They are able to deal with extended character data (such as Unicode) whereas the older *list* syntax cannot.

There are a number of GNU Awk (*gawk*) specific regular expression operators, some of which we touched on in the recap.

\s matches any whitespace character. Equivalent to the '[:space:]' character class in a *bracket expression* (i.e. '[[:space:]]').

\S matches any character that is not whitespace. Equivalent to '[^[:space:]]'.

\w matches any *word* character. A *word* character is any letter or digit or the underscore character.

\W matches any *non-word* character.

\< (*backslash less than*) matches the empty string at the beginning of a word.

\> (*backslash greater than*) matches the empty string at the end of a word.

\y (*backslash y*) matches a word boundary; that is it matches if the character to the left is a *word* character and the character to the right is a *non-word* character, or vice-versa.

\B Matches everywhere but on a word boundary; that is it matches if the character to the left and the character to the right are either both *word* characters or both *non-word* characters. This is essentially the opposite of '\y'.

\` (*backslash backquote*) matches the empty string at the beginning of a string. This is essentially the same as the '^' (*circumflex* or *caret*) operator, which means the beginning of the current line ($0), or the start of a string.

\' (*backslash single quote*) matches the empty string at the end of a string. This is essentially the same as the '$' (*dollar sign*) operator, which means the end of the current line ($0), or the end of a string.

GNU Awk can behave as if it is traditional Awk, or will operate only in POSIX mode or can turn on and off other regular expression features. There is a discussion of this in the GNU Awk manual, particularly in the Regular Expression section.

Functions

The details of the built-in functions we will be looking at here can be found in the GNU Manual in the *String-Manipulation Functions* section.

The `sub` function

The `sub` function has the format:

```
sub(regexp, replacement [, target])
```

The first argument *regexp* is a regular expression. This usually means it is enclosed in '//' delimiters[1].

The second argument *replacement* is a string to be used to replace the text matched by the *regexp*. If this contains a '&' character this refers to the text that was matched.

[1] This is a "Regexp Constant", but there is another form the "Computed Regexp", which is discussed in the GNU Manual.

The optional third argument *target* is the name of the string or field that will be changed by the function. It has to be an existing string variable or field since `sub` changes it in place. If the *target* is omitted then field '$0' (the whole input line) is modified.

The purpose of the `sub` function is to search the string in the *target* variable for the longest leftmost match with the *regexp* argument. This is replaced by the *replacement* argument.

The function returns the number of changes made (which can only be zero or 1).

Examples using `sub`

```
$ echo "banana" | awk '{sub(/an/,"XX"); print}'
bXXana
```

The first occurrence of the string 'an' is matched in the '$0' field, and replaced by 'XX'.

```
$ echo "banana" | awk '{sub(/an/,"&&"); print}'
bananana
```

This time the matched string is replaced by itself twice ('anan').

```
$ echo "banana" | awk '{n = sub(/an/,"&&"); print "Changes made=" n, "Result:", $0}'
Changes made=1 Result: bananana
```

Here the result of the `sub` function is stored in the variable `n` and it and the result are printed.

The `gsub` function

The `gsub` function is similar to `sub` and has the format:

```
gsub(regexp, replacement [, target])
```

As with `sub`, the arguments have the same purpose.

The function differs in that it searches *target* for **all** matches, and replaces them. The matches must not overlap (see below).

The function returns the number of changes made.

Examples using `gsub`

```
$ echo "banana" | awk '{gsub(/an/,"XX"); print}'
bXXXa
```

All occurrences of the string 'an' are matched in the '$0' field, and replaced by 'XX'.

```
$ echo "banana" | awk '{gsub(/ana/,"XX"); print}'
bXXna
```

Here there are two *overlapping* instances of 'ana', but only the first is replaced.

```
$ awk '{n = gsub(/[aeiou]/,"?",$1); printf "%-12s (%d)\n",$1,n}' file1.txt
n?m?         (2)
?ppl?        (2)
b?n?n?       (3)
str?wb?rry   (2)
gr?p?        (2)
?ppl?        (2)
pl?m         (1)
k?w?         (2)
p?t?t?       (3)
p?n??ppl?    (4)
```

This time we used the example file `file1.txt` and replaced all vowels with question marks, then captured the number changed. We printed the result and the number of changes.

The `gensub` function

This function is different from the other two, and has been added to GNU Awk later than `sub` and `gsub`[2]:

```
gensub(regexp, replacement, how [, target])
```

First argument: *regexp*

This is a regular expression (usually a *regexp constant* enclosed in slashes). Any of the regular expression operators seen in this and the last episode can be used. In particular, regular expressions enclosed in parentheses can be used here. (Similar features were described in the "Learning sed" series).

Second argument: *replacement*

In this argument, which is a string, the text to use for replacement is defined. This can also contain *back references* to text "captured" by the parenthesised expressions mentioned above.

The back references consist of a backslash followed by a number. If the number is zero then the it refers to the entire regular expression and is equivalent to the '&' character. Otherwise the number may be 1 to 9, referring to a parenthesised group.

[2] As a possible point of interest, I have a copy of the "*GAWK Manual*" (as it was called), dated 1992, version 0.14, which does not contain `gensub`.

Because of the way Awk processes strings, it is necessary to double the backslash in this argument. For instance, to refer to parenthesised component number one the string must be "\\1".

Third argument: *how*

This is a string which should contain 'G', 'g' or a number.

If 'G' or 'g' (*global*) it means that all matches should be replaced as specified.

If it is a number then it indicates which particular numbered match and replacement should be performed. It is not possible to perform multiple actions with this feature.

Fourth argument: *target*

If this optional argument is omitted then the field '$0' is used. Otherwise the argument can be a string, a variable (containing a string) or a field.

The *target* is **not changed** in situ, unlike with `sub` and `gsub`. The function returns the changed string instead.

Examples using `gensub`

```
$ echo "banana" | awk '{print gensub(/a/,"A","g"); print}'
bAnAnA
banana
```

Here `gensub` matches every occurrence of 'a', replacing it with capital 'A' globally. Note how we print the result of the `gensub` call. Note also that '$0' has not changed as can be seen when we print it with the second `print` statement.

```
$ echo "banana" | awk '{print gensub(/a/,"A","1")}'
bAnana
```

In this example we have requested that only the first match be replaced. There is no way to replace anything other than all matches or just one using the *how* argument.

```
$ echo "banana" | awk '{print gensub(/\Ba\B/,"A","g")}'
bAnAna
```

This example shows another way to replace matching letters. In this case we have specified only 'a's which are not at a word boundary. This is not an ideal solution.

```
$ echo "Hacker Public Radio" | awk '{print gensub(/(\w)(\w+)(\W*)/,"\\2\\1ay\\3","g")}'
ackerHay ublicPay adioRay
```

This example shows the use of regular expression groups and back references. The three groups are:

1. A single "word" character
2. One or more "word" characters
3. Zero or more non-"word" characters

Having matched these items (e.g. 'H', 'acker' and space for the first word), they are replaced by the second group ('acker'), the first group ('H'), the letters 'ay' and the third group (space). This is repeated throughout the *target*.

Since the *target* text consists of three words the regular expression matches three times (since argument *how* was a 'g') and the words are all processed the same way - into primitive "Pig Latin".

```
$ awk 'BEGIN{print gensub(/(\w)(\w+)(\W*)/,"\\2\\1ay\\3","3","Hacker Public Radio")}'
Hacker Public adioRay
```

This example is a variant of the previous one. In this case the entire Awk script is in a 'BEGIN' rule, and the *target* is a string constant. Since argument *how* is the number 3 then only the third match is replaced.

Example script

I have included a longer example using a new test datafile. The example Awk script is called `contacts.awk` and the data file is `contacts.txt`. They are included with this show and links to them are listed below.

The test data was generated on a site called "Mockaroo", which was used to generate CSV data. The Vim plugin `csv.vim` was used to reformat this into the final format with the `:ConvertData` function. Here are the first 8 lines from that file:

```
Name: Robin Richardson
First: Robin
Last: Richardson
Email: rrichardson0@163.com
Gender: Female

Name: Anne Price
First: Anne
```

Here is the entire awk script which can be run thus:

```
awk -f contacts.awk contacts.txt
```

```
1  #!/usr/bin/awk -f
2
3  #
4  # Define separators
```

```
#
BEGIN{
    #
    # The field separator is a newline
    #
    FS = "\n"

    #
    # The record separator is two newlines since there's a blank line between
    # contacts.
    #
    RS = "\n\n"

    #
    # On output write a line of hyphens on a new line
    #
    ORS = "\n----\n"
}

{
    #
    # Show where the "beginning of buffer" is
    #
    sub(/\`/, "[")

    #
    # Show where the "end of buffer" is
    #
    sub(/\'/, "]")

    #
    # Show where the start and end of "line" are
    #
    sub(/^/, "{")
    sub(/$/, "}")

    #
    # Print the buffer with a record number and a field count
    #
    print "(" NR "/" NF ")", $0
}
```

The script changes the default separators in order to treat the entire block of lines making up a contact as a single Awk "record". Each field is separated from the next with a newline, and each "record" is separated from the next by two newlines. For variety when printing the output "records" are separated by a

newline, four hyphens and a newline.

As it processes each "record" the script marks the positions of four boundaries using some of the regular expression operators we have seen in this episode. It prints the "record" ($0) preceding it by the record number and the number of fields.

A sample of the first 8 lines of the output looks like this:

```
(1/5) {[Name: Robin Richardson
First: Robin
Last: Richardson
Email: rrichardson0@163.com
Gender: Female]}
----
(2/5) {[Name: Anne Price
First: Anne
```

Warning for `sed` users

GNU awk is related to GNU sed, which was covered in the series "Learning sed". If you listened to that series there is unfortunately some potential for confusion as we learn about GNU Awk. Many of the regular expression operators described for GNU sed are the same as those used in GNU Awk **except** that sed uses a backslash in front of some and Awk does not. Examples are '\+' and '\?' in sed versus '+' and '?' in Awk.

Summary

This episode covered:

- A recap of the last episode
  - Correcting some small errors in the list of regular expression operators
- More detail of regular expression operators
- A detailed description of the functions `sub`, `gsub` and `gensub` with examples
- A more complex example Awk script
- A warning about the differences in regular expressions between sed and Awk

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
- The "*Learning sed*" series:
  - "Introduction to sed - part 1" - episode 1976
  - "Introduction to sed - part 2" - episode 1986
  - "Introduction to sed - part 3" - episode 1997
  - "Introduction to sed - part 4" - episode 2011
  - "Introduction to sed - part 5" - episode 2060
- The "Mockaroo" data generator site
- The Vim plugin "csv.vim"
- Resources:
  - ePub version of these notes
  - PDF version of these notes
  - Demonstration of some regex operators: contacts.awk
  - File of dummy contacts: contacts.txt