Chapter 5

Processing tabulated data

Chapter 5 of this course will discuss a tool called "`awk`"[1] that is suitable for working with tabular data such as can be found in the file `table` in the `awk` demo directory, `/info/courses/text.tools.course/awk-examples`.

```
$ cd awk

$ cat table
Archaeology     22      44
Biotechnology   2       7
CS              71      138
CUS             139     1204
...
```

Similarly to `sed`, `awk` goes through its input one line at a time, applies a test to that input line and if the test succeeds performs the action later on the command line. A missing test statement, as with `sed`, implies that the action is applied to all lines. Here are some simple examples to illustrate the general shape of `awk` commands:

```
$ awk '{print $1}' < table
Archaeology
Biotechnology
CS
CUS
...
```

This example illustrates several points about `awk` commands. They don't need the "`-e`" key that `sed` requires (though older versions of `awk` *do*), the actions are surrounded by curly brackets, or "braces", and for each line the first column's contents can be addressed as "`$1`". There is no test performed before the action, so it applies to all lines. Also note that this operation could have been performed just as well by the `cut` command. (See its `man` page for details.)

[1] `awk` is named after the initials of its creators: A.V. *A*ho, P.J. *W*einberger and B.W. *K*ernigan.

```
$ awk '$2>20 {print $1}' < table
Archaeology
CS
CUS
Chemistry
Materials
Thor
```

This example illustrates how tests can be added to **awk**'s commands. The test "$2>20" checks to see if the contents of column 2 are greater than 20 and if they are then the action is performed.

The general syntax of an awk command is "*test* {*action*}".

Exercise 24: Print out the first column of all the lines whose third column is greater than 100.

5.1 Printing from awk

We will start with some very simple **awk** printing actions that are often used without any corresponding tests. Remember that the curly brackets around the action are needed even when there is no test to perform.

5.1.1 Simple printing of selected columns

As **awk** processes each line in turn, it splits the line into columns. These are separated by default by whitespace in the input file. In the action, and in the test, the contents of these columns can be refered to as "$1", "$2", "$3" *etc.* So, to print the second column of a file:

```
$ awk '{print $2}' < table
22
2
71
139
...
```

If we want to print more than one column then the corresponding dollar terms should be separated by commas. If they are not then the columns are printed without any separation:

```
$ awk '{print $1, $2}' < table
Archaeology 22
Biotechnology 2
CS 71
CUS 139
...
```

```
$ awk '{print $1 $2}' < table
Archaeology22
Biotechnology2
CS71
CUS139
...
```

Literal strings can also be printed:

```
$ awk '{print $1 "," $2}' < table
Archaeology,22
Biotechnology,2
CS,71
CUS,139
...
```

The dollar expression "`$0`" has special meaning and means the entire line:

```
$ awk '{print $0}' < table
Archaeology     22      44
Biotechnology   2       7
CS              71      138
CUS             139     1204
...
```

The command "`print`" on its own is equivalent to "`print $0`", *i.e.* to print the entire line.

Exercise 25: Print the (entire) lines from `table` where the third column has a value greater than 100.

It is possible to mix literal strings with the columns' contents in a `print` command:

```
$ awk '{print $1, ":", $2}' < table
Archaeology : 22
Biotechnology : 2
CS : 71
CUS : 139
...
```

5.1.2 Formatted printing of selected columns

To perform formatted printing, `awk` provides a `printf` action with the same formatting facilities as `C`'s library call of the same name:

```
$ awk '{printf "%-16s% 4d\n", $1, $2}' < table
Archaeology       22
Biotechnology      2
CS                71
CUS              139
...
```

The "\n" at the end of the formatting string represents the newline character that marks the end of a line. `printf` will not end a line at the end of its output unless explicitly instructed to in this way; `print` always does.

Integers: These can be formatted with the "%d" formatting directive.

```
$ awk '{printf "%d\n", $2}' < table
22
2
71
139
...
```

The amount of space (given as the number of characters) to provide for each integer can be specified between the "%" and the "d":

```
$ awk '{printf "%3d\n", $2}' < table
 22
  2
 71
139
...
```

To have the padding done with zeroes rather than spaces, replace the "3" with "03":

```
$ awk '{printf "%03d\n", $2}' < table
022
002
071
139
...
```

To have the sign inserted, even when it is positive, precede the "width" parameter with a "+". You need to increase the width parameter by one to allow for this extra character:

```
$ awk '{printf "%+4d\n", $2}' < table
 +22
  +2
 +71
+139
...
```

Without a width parameter, the sign is simply prepended to the number:

```
$ awk '{printf "%+d\n", $2}' < table
+22
+2
+71
+139
...
```

If inadequate room is provided for an integer then it will overrun:

```
$ awk '{printf "%+03d\n", $2}' < table
+22
+02
+71
+139
...
```

Exercise 26: Write an `awk printf` command to generate the following output from `table`:

```
␣␣22␣␣␣44
␣␣␣2␣␣␣␣7
␣␣71␣␣138
␣139␣1204
␣␣50␣␣102
␣␣␣4␣␣␣13
␣␣15␣␣␣31
␣␣65␣␣131
␣␣␣4␣␣␣␣9
␣␣␣8␣␣␣16
␣111␣␣202
```

where "␣" indicates a single space.

Floating point numbers: These can be treated similarly, but now there are two "width" parameters to specify if wanted: one for the digits before the decimal point and one for the whole number. The equivalent character to "`d`" for floating point numbers is "`f`". `awk` has a built-in default to print six digits after the decimal point:

```
$ awk '{printf "%f\n", $2}' < table
22.000000
2.000000
71.000000
139.000000
...
```

The number of decimal places to show can be specified by following the "`%`" with a decimal point and the number wanted. So, for example, to have three decimal places:

```
$ awk '{printf "%.3f\n", $2}' < table
22.000
2.000
71.000
139.000
...
```

To provide a certain number of columns for the number to be printed in, place the total width of the number before the decimal point in the format statement. *n.b.* this is the *total* width of the string printed, not the number of digits before the decimal point. To further complicate matters, the total width must allow for the space for the sign character (even if it is just a space, which it is by default for positive numbers).

```
$ awk '{printf "%8.3f\n", $2}' < table
  22.000
   2.000
  71.000
 139.000
...
```

To cause the sign to be printed, even when it is positive, place a "+" after the "%":

```
$ awk '{printf "%+8.3f\n", $2}' < table
 +22.000
  +2.000
 +71.000
+139.000
...
```

Again, if insufficient space is allocated, numbers will overrun:

```
$ awk '{printf "%+7.3f\n", $2}' < table
+22.000
 +2.000
+71.000
+139.000
...
```

Exercise 27: Generate the following output from `table`:

```
␣␣22␣␣␣44␣␣␣␣2.000
␣␣␣2␣␣␣␣7␣␣␣␣3.500
␣␣71␣␣138␣␣␣␣1.944
␣139␣1204␣␣␣␣8.662
␣␣50␣␣102␣␣␣␣2.040
␣␣␣4␣␣␣13␣␣␣␣3.250
␣␣15␣␣␣31␣␣␣␣2.067
␣␣65␣␣131␣␣␣␣2.015
␣␣␣4␣␣␣␣9␣␣␣␣2.250
␣␣␣8␣␣␣16␣␣␣␣2.000
␣111␣␣202␣␣␣␣1.820
```

(Hint: "`{printf "%f\n", $3/$2}`" is valid syntax for printing column three divided by column two.)

Strings: These are handled with the "%s" format directive and its variants. In its simplest form it prints a string:

```
$ awk '{printf "%s\n", $1}' < table
Archaeology
Biotechnology
CS
CUS
...
```

Specifying a width for this field involves placing the width between the "%" and the "s" as before. By default this right-aligns the strings:

```
$ awk '{printf "%16s\n", $1}' < table
     Archaeology
   Biotechnology
              CS
             CUS
...
```

If a string is too long for the field it is put in, it will overrun:

```
$ awk '{printf "%10s\n", $1}' < table
Archaeology
Biotechnology
        CS
       CUS
...
```

To prevent overrunning, it is possible to specify the maximum number of characters to be printed from a string. To do this, use the floating point "precision" directive of a decimal point and a number:

```
$ awk '{printf "%16.10s\n", $1}' < table
      Archaeolog
      Biotechnol
              CS
             CUS
...
```

In this example, 16 columns are allocated for the strings but only 10 or fewer characters are printed.

To have a string left-aligned in its allocated set of columns, precede the width parameter with a "-":

```
$ awk '{printf "%-10.10s\n", $1}' < table
Archaeolog
Biotechnol
CS
CUS
...
```

It is also possible, though rarely useful, to truncate strings to a certain maximum number of characters but not to allocate a minimum number of columns:

```
$ awk '{printf "%.10s %d\n", $1, $2}' < table
Archaeolog 22
Biotechnol 2
CS 71
CUS 139
...
```

Literal Strings: Literal strings can be placed in the format statement. If a literal percentage character is needed use "%%" and if a backslash is needed use "\\":

```
$ awk '{printf "Dept:%s %% %d\n", $1, $2}' < table
Dept:Archaeology % 22
Dept:Biotechnology % 2
Dept:CS % 71
Dept:CUS % 139
...
```

5.2 awk scripts

Before we proceed to the more elaborate uses of `awk` we will need to understand `awk` scripts. Just as it is possible to put sets of `sed` instructions into a file and pass the file to `sed` with the "`-f`" key, it is possible to do the same with `awk`:

```
$ ls -l awkfile
-rw-r--r--   1 rjd4     uxsup         39 Mar 20 15:13 awkfile

$ cat awkfile
# This line is a comment
#
{print $1}

$ awk -f awkfile < table
Archaeology
Biotechnology
CS
CUS
...
```

(Again, just the first four lines of output are shown to save space.) Note that, just like `sed`, `awk` has a "`-f`" key to introduce a command file, "`#`" acts as a comment character and blank lines are ignored.

Following the mechanism we saw with `sed`, it is possible to create `awk` scripts as well:

```
$ ls -l awkscript1
-rwxr-xr-x   1 rjd4     uxsup          53 Mar 20 15:16 awkscript1

$ cat awkscript1
#!/bin/awk -f
# This line is a comment

{print $1}

$ ./awkscript1 < table
Archaeology
Biotechnology
CS
CUS
...
```

(Remember to make the awk command executable as well as readable.) We will use awk scripts to introduce multiple awk commands. As an example, consider:

```
$ ls -l awkscript2
-rwxr-xr-x   1 rjd4     uxsup          75 Mar 20 15:18 awkscript2

$ cat awkscript2
#!/bin/awk -f
{printf "%-16s", $1}
{printf "%8d", $2}
{printf "%8d\n", $3}

$ ./awkscript2 < table
Archaeology           22      44
Biotechnology          2       7
CS                    71     138
CUS                  139    1204
...
```

An action can be spread over several lines, but must still remain within the pair of brackets:

```
$ ls -l awkscript3
-rwxr-xr-x   1 rjd4     uxsup          71 Mar 20 17:17 awkscript3

$ cat awkscript3
#!/bin/awk -f
{printf "%-16s", $1
 printf "%8d", $2
 printf "%8d\n", $3}

$ ./awkscript3 < table
Archaeology           22      44
Biotechnology          2       7
CS                    71     138
CUS                  139    1204
...
```

Multipart actions can be condensed to a single line with the parts separated by semicolons instead of newlines:

```
$ ls -l awkscript4
-rwxr-xr-x   1 rjd4     uxsup         73 Mar 20 17:20 awkscript4

$ cat awkscript4
#!/bin/awk -f
{printf "%-16s", $1; printf "%8d", $2; printf "%8d\n", $3}

$ ./awkscript4 < table
Archaeology           22      44
Biotechnology          2       7
CS                    71     138
CUS                  139    1204
...
```

5.3 awk's tests

A test statement precedes the bracketed action statement. It has access to the columns of the line under consideration in just the same way as the actions through the use of "$1", "$2" *etc.*

5.3.1 Numerical comparisons

Given columns with numerical data in them, a variety of numerical comparisons can be performed:

```
$ awk '$3>200 {print $1}' < table
CUS
Thor
```

The following numerical comparisons are available:

==	equal to	!=	not equal to
>	strictly greater than	<	strictly less than
>=	greater than or equal to	<=	less than or equal to

Note that the equality test uses a *double* equals sign. (A single equals sign, as we will see later, is an assignment operator.)

5.3.2 String comparisons

Given columns with string data in them, a variety of string tests can be performed. The most useful are equality testing and regular expression matching.

Equality testing: This is done by using "==" again:

```
$ awk '$1=="CUS" {print $1, $2, $3}' < table
CUS 139 1204
```

Note that the literal string is surrounded by double quotation characters.

It is also possible to apply the other five tests seen in the numerical context above. The standard ASCII lexicographic ordering is used. See "`man ascii`" for a list of the ASCII characters in numerical order.

```
$ awk '$1<"CUS" {print $1, $2, $3}' < table
Archaeology 22 44
Biotechnology 2 7
CS 71 138
```

Regular expression matching: This is done by replacing the equality test, "==", with the tilde character, "~", and the double quotation marks around the literal string each with a slash, "/":

```
$ awk '$1~/^C/ {print $1, $2, $3}' < table
CS 71 138
CUS 139 1204
Chemistry 50 102
```

Unfortunately, `awk`'s regular expressions are based on `egrep`, rather than the typically more useful `grep`.

If a regular expression is given on its own, without anything to explicitly match it against any column, then it is matched against the whole line. In other words, the test "/*reg.exp.*/" is equivalent to "`$0~`/*reg.exp.*/". For example:

```
$ awk '/^C/ {print $0}' < table
CS              71      138
CUS             139     1204
Chemistry       50      102

$ awk '$0~/^C/ {print $0}' < table
CS              71      138
CUS             139     1204
Chemistry       50      102
```

5.3.3 Matching line numbers

`awk` maintains a special variable "`NR`" ("*N*umber of *R*ecord") that keeps track of the line number of the input. A particular line of the input can be selected by testing this variable:

```
$ awk 'NR==2 {print $0}' < table
Biotechnology   2       7
```

Note that, unlike the shell, no dollar is required to "expand" `NR`.

In addition, `awk` has two special line descriptors called "`BEGIN`" and "`END`". These match fictitious, empty lines that occur before and after the actual input. For example:

```
$ ls -l awkscript5
-rwxr-xr-x   1 rjd4     uxsup          74 Apr  3 13:40 awkscript5

$ cat awkscript5
#!/bin/awk -f
BEGIN   {print "BEGINNING"}
        {print $1}
END     {print "ENDING"}

$ ./awkscript5 < table
BEGINNING
Archaeology
Biotechnology
CS
CUS
...
ENDING
```

Exercise 28: Write an `awk` command to print each line of a file preceded by its line number in the same format as is provided by "`cat -n`". (See `cat`'s `man` page for details.)

5.3.4 Pairs of line identifiers

Again as with `sed`, it is possible to give a comma-separated pair of line descriptors which define a range of lines (possibly more than one range in the case of regular expressions, remember). For example:

```
$ awk 'NR==1,NR==4 {print $0}' < table
Archaeology     22      44
Biotechnology   2       7
CS              71      138
CUS             139     1204
```

5.4 Combining tests

It is possible to make combination tests in `awk` using the operators "`||`" for "or", "`&&`" for "and" and "`!`" for "not":

```
$ awk '$2>100 && $3>100 {print}' < table
CUS             139     1204
Thor            111     202

$ awk '$2>100 && ! $3>1000 {print}' < table
Thor            111     202
```

Tests may be placed inside brackets. This is useful for combining them and not having to worry about what "`A && B || C`" means. (It means "`(A && B) || C`".)

```
$ awk '($2>100 && $3>100) || ($1~/^C/) {print}' < table
CS              71      138
CUS             139     1204
Chemistry       50      102
Thor            111     202

$ awk '($2>100) && ($3>100 || $1~/^C/) {print}' < table
CUS             139     1204
Thor            111     202
```

Exercise 29: Find all the departments in `table` that start with a vowel and that sent more than 10 messages (column 2 is greater than 10).

5.5 User-defined variables

We have already seen some of which `awk` provides a variable, "`NR`", which it updates automatically. There are many more besides these that we will meet later. In this section, however, we will introduce variables that the user can use.

5.5.1 Simple variables

A variable can be defined simply by referring to it. There is no need to "declare" it. Here is a simple example:

```
$ ls -l awkscript6
-rwxr-xr-x   1 rjd4     uxsup         62 Mar 20 16:14 awkscript6

$ cat awkscript6
#!/bin/awk -f
        {sum = sum + $2}
END     {print sum}

$ ./awkscript6 < table
491
```

For every line in the input the action "`sum = sum + $2`" is performed (because there is no test preceding it, it is applied to every line). The first time this action is used, a variable `sum` is created and initialized to 0. Each time this action is used, this action adds the contents of the second column to the value of `sum`. Finally, after the last line of the input has been processed the "`END`" command line is applied. Its action is to print the value of `sum`.

There is a short-cut available for operations like "`sum = sum + $2`". This expression can be written as "`sum += $2`". There are analogous expressions, "`-=`", "`*=`" and "`/=`" for the other three basic operations.

Here's a more typical example of an `awk` script. The table of data in the file `table` is actually a list of departments, the number of mail messages I received from them in a month and the total amount of space taken up by these messages. Suppose I wanted the mean size of a messages for each department, the total number of messages, total amount of disc space, mean number of messages, mean amount of space taken and mean size of a message overall. Here's a simple `awk` script to do it:

```
$ cat means
#!/bin/awk -f

# Print some column headings and initialize variables
BEGIN   {
        printf "%-16s %8s %8s %8s\n", "Department", "No.", "Space", "Mean"
        }

# For each line, print the three columns, the mean message size, and
# add to sum2 and sum3
        {
        printf "%-16s %8d %8d %8.2f\n", $1, $2, $3, $3/$2
        sum2 += $2; sum3 += $3
        }

# Print the summaries at the end:
END     {
        printf "%-16s %8d %8d %8.2f\n", "TOTALS:", sum2, sum3, sum3/sum2
        printf "%-16s %8.2f %8.2f\n", "MEANS:", sum2/NR, sum3/NR
        }
```

Note how the use of the length parameters in the `printf` format directives ensures that the columns align with the headings and the summaries.

```
$ ./means < table
Department            No.    Space     Mean
Archaeology            22       44     2.00
Biotechnology           2        7     3.50
CS                     71      138     1.94
CUS                   139     1204     8.66
...
TOTALS:               491     1897     3.86
MEANS:              44.64   172.45
```

5.5.2 Column variables

As well as defining new variables, it is also possible to redefine the column ("dollar") variables. For example, to limit the values in column 2 to 100, we might try the following:

```
$ awk '$2>100 {$2=100}; {print}' < table
Archaeology     22      44
Biotechnology   2       7
CS              71      138
CUS 100 1204
Chemistry       50      102
...
Physics         8       16
Thor 100 202
```

Note that the two lines affected, the "CUS" and "Thor" lines, have lost their formatting. If any columns are altered **awk** loses the information about what whitespace there was between them. Therefore, if the contents of columns are going to be redefined, "printf" is to be preferred over "print":

```
$ awk '$2>100 {$2=100}; {printf "%-14s %4d %4d\n", $1, $2, $3}' < table
Archaeology      22   44
Biotechnology     2    7
CS               71  138
CUS             100 1204
Chemistry        50  102
...
Physics           8   16
Thor            100  202
```

5.5.3 Array variables

The variables sum2 and sum3 are "scalar" variables. It is possible to create an "array" variable that holds a collection of values. The particular value is selected by a key called an "index". Consider this simplified version of of the "means" program:

```
$ cat means1
#!/bin/awk -f
        { sum2 += $2; sum3 += $3 }
END     { printf "%-16s%8d%8d\n", "TOTALS:", sum2, sum3 }

$ ./means1 < table
TOTALS:              491    1897
```

In this example sum2 and sum3 are simple, scalar variables. We could also write it like this:

```
$ cat means2
#!/bin/awk -f
        { sum[2] += $2; sum[3] += $3 }
END     { printf "%-16s%8d%8d\n", "TOTALS:", sum[2], sum[3] }

$ ./means2 < table
TOTALS:              491    1897
```

In this example `sum` is an array variable with two values, indexed by 2 and 3. This has gained us nothing in this case except the confusion of new notation, but it paves the way for more general facilities later.

A particularly important array is "`$`". This is special in that it does not (and cannot) use the square brackets around its index. To demonstrate that `$` really is an array variable, we can replace the number that follows it ("1", "2" or "3" in our examples) with a variable that has one of these values. For example:

```
$ cat dollar1
#!/bin/awk -f
BEGIN   {i=1}
        { print $i }

$ ./dollar1 < table
Archaeology
Biotechnology
CS
CUS
...
```

So we can use an expression "`$i`" for "`$1`" if the variable `i` has been given the value 1 previously.

Giving `i` a static value is fairly pointless, however, so we will now introduce the instructions for making it run through a loop. Consider the following example carefully:

```
$ cat loop1
#!/bin/awk -f
        {
        for ( i = 3; i >= 1; i -= 1 ) printf "%s ", $i
        printf "\n"
        }


$ ./loop1 < table
44 22 Archaeology
7 2 Biotechnology
138 71 CS
1204 139 CUS
...
```

So here we are able to reverse the order of the fields in each line by causing the variable `i` to run down from 3 to 1. The `for` command runs the `printf` instruction three times, once for each of the values of `i`. This is called "looping".

The `for` instruction has the structure "`for (`*initialize*`;` *test*`;` *increment*`)` *instruction*". The "*initialize*" phase simply sets up the value of a loop variable at the start. In this case "`i = 3`" defines `i` and sets it to be 3. The "*test*" phase provides some sort of test, often on the value of the loop variable, but not necessarily. This test is performed just before the start of the command to be repeated. If the test is true then the command is performed. If the test is false, the `for` loop is abandoned. In our example, the test is "`i >= 1`" which tests to see that `i` does not fall below 1. Next comes the "*increment*" phase. This is applied immediately after each repeat of the command and gives `i` its next value. In our example, "`i -= 1`", the variable `i` is decremented by 1 after each run of the instruction. Finally, after the closing round bracket of the `for` statement, comes the instruction to be run. In our case it is "`printf "%s ", $i`". This is called three times with `i` taking the values 3, 2 and 1.

We can now write a more general version of the `loop1` example. `awk` has an internal variable "`NF`" ("*N*umber of *F*ields") giving the number of columns on the current line. The more general `awk` program, `loop2`, given below will reverse the columns of a data file with arbitrarily many columns:

```
$ cat loop2
#!/bin/awk -f
        {
        for ( i=NF; i >= 1; i -= 1 ) printf "%s ", $i
        printf "\n"
        }
```

```
$ ./loop2 < table
44 22 Archaeology
7 2 Biotechnology
138 71 CS
1204 139 CUS
102 50 Chemistry
13 4 Engineering
31 15 Geography
131 65 Materials
9 4 OldSchools
16 8 Physics
202 111 Thor
```

Unfortunately, the `NF` variable in an `END` statement takes the value 0. As a consequence, some care is needed to work with tables of data of an unknown number of columns. Here is an example of a workaround for this problem; with each line of input, the value of `NF` is compared against a variable `max` which is kept equal to the largest value of `NF`. `max` can then be used in the `END` statement:

```
$ cat means3
#!/bin/awk -f
NF>max  { max=NF }
        { for(i=1;i<=NF;i++)
                {
                sum[i]+=$i
                printf "%16s", $i
                }
        }
        { printf "\n" }
END     {
         for(i=1;i<=max;i++) printf "%16.3f", sum[i]/NR
         printf "\n"
        }


$ ./means3 < table
     Archaeology              22              44
   Biotechnology               2               7
              CS              71             138
             CUS             139            1204
...
           0.000          44.636         172.455
```

This also shows that if a string (column 1) is used as a number it takes the value 0 and that multiple commands can be looped by a `for` statement by placing them in curly brackets.

All normal arrays in `awk` (all of them except `$`), are "associative arrays". This means that the index can be an arbitrary value, not just a positive integer. Suppose we didn't have just one table of figures, but several, with departments occuring in more than one of the tables:

```
$ cat table2
Archaeology     25      47
CS              107     206
Engineering     5       13
Pharmacology    1       2
Physics         17      34


$ cat summer1
#!/bin/awk -f
        {
        number[$1] += $2
        size[$1] += $3
        }
END     {
        print "Physics", number["Physics"], size["Physics"]
        }

$ cat table table2 | ./summer1
Physics 25 50
```

So we can use strings (either the `$1` string or the literal string "`Physics`" in our example) as an index to `number` and `size`.

Of course, it is of limited use if we have to rewrite and rerun the scripts for each department. Clearly we would like to be able to output all the departments that are keys in the `number` and `size` arrays. This can be done by modifying the `for` syntax in a particular way, as illustrated by this example:

```
$ cat summer2
#!/bin/awk -f
        {
        number[$1] += $2
        size[$1] += $3
        }
END     {
        for ( dept in number ) print dept, number[dept], size[dept]
        }


$ cat table table2 | ./summer2
CS 178 344
Chemistry 50 102
...
CUS 139 1204
Physics 25 50
```

The `for` loop this time does not follow the "*initialize i; test i; increment i*" pattern seen before but instead has the form "*key* `in` *array*". The `awk` instruction after the `for` statement (a `print` statement in this case) is run once for each key in the array `number`. Each time it is run the key `dept` has a different value. Note that the order the keys appear in is random. It is neither alphabetic order nor the order in which the keys were first encountered by the script. Typically, the output of a program like this would be piped through `sort` to yield a known order:

```
$ cat table table2 | ./summer2 | sort
Archaeology 47 91
Biotechnology 2 7
...
Pharmacology 1 2
Physics 25 50
Thor 111 202
```

5.6 Column delimiters

So far in this part of the course we have glibly refered to "columns" as if their meaning was obvious. The fields in each line forming the columns have been separated by white space. In our running example, I have cheated by compressing the name of "Old Schools" into a string "`OldSchools`" without spaces. Other separators are possible and often necessary. After all, what would we do if the text columns we wanted to use needed spaces in the entries? To solve this problem, `awk` has an internal variable called "`FS`" (*F*ield *S*eparator) which defines the character that is used to separate columns.

For example consider the file `/etc/passwd`. This carries the information about the users of a Unix system and consist of a number of lines of the form

"`rjd4:abcdefghijlmn:2049:20:Bob Dowling:/home/rjd4:/bin/bash`".

The fields in this file's lines are separated by colons. `awk` could handle a file like this in the following manner:

```
$ cat passwd1
#!/bin/awk -f
BEGIN   {FS=":"}
{printf "%s:%s\n", $1, $5}


$ ./passwd1 < /etc/passwd
root:0000-Admin(0000)
daemon:0000-Admin(0000)
bin:0000-Admin(0000)
sys:0000-Admin(0000)
...
```

If `awk` is being used to generate one table from another, it can also be convenient to define the *output* column delimiter. This is defined by the variable "`OFS`" and is used by `print` to separate its output fields:

```
$ cat passwd2
#!/bin/awk -f
BEGIN   {FS=":"; OFS=":"}
{print $1, $5}


$ ./passwd2 < /etc/passwd
root:0000-Admin(0000)
daemon:0000-Admin(0000)
bin:0000-Admin(0000)
sys:0000-Admin(0000)
...
```

a==*b*	a is equal to *b*
a!=*b*	a is not equal to *b*
a>*b*	a is strictly greater than *b*
a<*b*	a is strictly less than *b*
a>=*b*	a is greater than or equal to *b*
a<=*b*	a is less than or equal to *b*
a~/*reg. exp.*/	a matches the regular expression
/*reg. exp.*/	the line (`$0`) matches the regular expression
`BEGIN`	before the first line of input
`END`	after the last line of input

Table 5.1: Summary of `awk` line tests

`NF`	*N*umber of *F*ields (columns) in the current line
`FS`	*F*ield *S*eparator — the character delimiting columns
`OFS`	*O*utput *F*ield *S*eparator — the character between output columns created by `print`
`NR`	*N*umber of *R*ecords — line number

Table 5.2: Summary of `awk` built-in variables