The awk programming

Session Objectives

- Introduction
- History of awk and gawk
- Running an awk program
- Referencing Fields in a line
- Constants and Variables
- Built in Variables
- Operators and Expressions:
- Statements and Control structures
- Regular Expressions
- Arrays in ‘Awk’
- Functions in Awk
- Shell and Awk

INTRODUCTION

If you are like many computer users, you would frequently like to make changes in various text files wherever certain patterns appear, or extract data from parts of certain lines while discarding the rest. To write a program to do this in a language such as C or Pascal is a time-consuming inconvenience that may take many lines of code. The job may be easier with awk.

Awk is a programming language designed to search for, match patterns, and perform actions on files. Awk programs are generally quite small, and are interpreted. The awk utility interprets a special-purpose programming language that makes it possible to handle simple data-reformatting jobs with just a few lines of code. This makes it a good language for prototyping. The basic function of awk is to search files for lines (or other units of text) that contain certain patterns. When a line matches one of the patterns, awk performs specified actions on that line. Awk keeps processing input lines in this way until the end of the input files are reached.

Programs in awk are different from programs in most other languages, because awk programs are *data-driven*; that is, you describe the data you wish to work with and then what to do when you find it. Most other languages are *procedural*; you have to describe, in great detail, every step the program is to take. When working with procedural languages, it is usually much harder to clearly describe the data your program will process. For this reason, awk programs are often refreshingly easy to both write and read.

The GNU implementation of awk is called gawk; it is fully upward compatible with the System V Release 4 version of awk. gawk is also upward compatible with the POSIX specification of the awk language. This means that all properly written awk programs should work with gawk. Thus, we usually don't distinguish between gawk and other awk implementations.

Using awk you can:

- manage small, personal databases
- generate reports
- validate data
- produce indexes, and perform other document preparation tasks
- even experiment with algorithms that can be adapted later to other computer languages

History of awk and gawk

The name awk comes from the initials of its designers: Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan. The original version of awk was written in 1977 at AT&T Bell Laboratories. In 1985 a new version made the programming language more powerful, introducing user-defined functions, multiple input streams, and computed regular expressions. This new version became generally available with Unix System V Release 3.1. The version in System V Release 4 added some new features and also cleaned up the behavior in some of the "dark corners" of the language. The specification for awk in the POSIX Command Language and Utilities standard further clarified the language based on feedback from both the gawk designers, and the original Bell Labs awk designers. The GNU implementation, gawk, was written in 1986 by Paul Rubin and Jay Fenlason, with advice from Richard Stallman. John Woods contributed parts of the code as well. In 1988 and 1989, David Trueman, with help from Arnold Robbins, thoroughly reworked gawk for compatibility with the newer awk. Current development focuses on bug fixes, performance improvements, standards compliance, and occasionally, new features.

Data File for the Examples

Many of the examples in this session take their input from a sample data file. The file, called `Student', represents a list of student names along with their marks. Each line in the file is considered to be one *record*.

$cat Student

```
Viji   90 85 98 100
Meera  90 90 90 100
Ann    98 98 76 97
Vasu   89 99 91 87
Sai    98 87 97 96
Hema   98 87 76 97
```

Running an awk program

'awk' can be invoked from the command line as follows

awk [-Fseparator] 'pattern {action}' data file

Awk looks at the input provided as a sequence of records where each record is a sequence of fields. The default record separator is the newline character and the default field separator is one or more bland spaces. If the field separator is not default, one of the ways of informing 'awk' is by using the '-F' option

Consider the file "Student" which is given earlier.

Example

$awk Viji Student
This will not produce any result, as awk is not instructed with any operation to perform on the file.

The correct command would be
$ awk /Viji/ Student
```
Viji   90 85 98 100
```
Here /Viji/ is a pattern understood by awk.

If no pattern is specified, then awk performs on all lines.
$awk {print} Student
```
Viji   90 85 98 100
Meera  90 90 90 100
Ann    98 98 76 97
Vasu   89 99 91 87
Sai    98 87 97 96
Hema   98 87 76 97
```


Referencing Fields in a line

One of the things that 'awk' can do is it can operate on individual fields within a line or record. 'Awk' considers every record in a file to be composed of 'fields'. Fields are separated by 'field separators', which are normally spaces or tabs. But the field separators can be changed to whatever delimiter or separator the user wishes to have.

Fields are accessed by the notation $n where n is the number of the field. The *first* field is referenced as $1 and so on. The field reference $0 refers to the full record.

To print the first and third record of the file Student, the command is

Example

```
$ awk '/Vasu/  {print $1 "   -  " $3}' Student
Vasu   -  89
```

Constants and Variables

Constants

Constants are of two types numeric and string. String constants are enclosed within double quotes.

User Defined Variables

Identifiers represent variables. Variable names can consist of alphabets, digits and underscore subject to the condition that variable names may not begin with a digit. Awk is case sensitive.

Examples:

- _xyz
- Xyz
- _930

Variables need not be declared before use. They are created when they are referenced for the first time. Variables have preferred type. But the interpretation of the variable depends on the context.

X="12" + 1

Here X is interpreted as a numeric value. The following statement concatenates and stores the result in another variable. Here name is a variable that stores the concatenated value of "Mr/Mrs." with the first field.

Example

```
$awk '{name="Mr/Mrs." $1;print name}' Student
Mr/Mrs.Viji
Mr/Mrs.Meera
Mr/Mrs.Ann
Mr/Mrs.Vasu
```



```
Mr/Mrs.Sai
Mr/Mrs.Hema
```

Built in Variables:

'awk' recognizes some built in variables that help to identity records, no of rows and so on.

The in built variables are:

Built-in Variable	Value
RS	Record Separator
FS	Field Separator
NR	Number of the current record being parsed
$0	Represents the current record
$1,$n	Represents the individual fields.

Operators and Expressions:

There are three classes of operators in 'awk'.

1. Arithmetic operators
2. Relational operators
3. Logical operators

Arithmetic operators

- +
- -
- *
- /
- %
- ++
- --
- =,+=,*=,/=,%=

Relational operators:

- ==
- !=
- >
- <
- >=
- <=

Logical Operators

- && AND
- || OR
- ! NOT

Example

$ awk ' $1 == "Meera"' Student (or) $awk ' $1 ~ "Meera"' Student
Meera 90 90 90 100

$ awk ' $2>90' Student
Ann 98 98 76 97
Sai 98 87 97 96
Hema 98 87 76 97

Statements and Control structures

print statement:

The 'print' statement can take in variable number of heterogeneous parameters and display them.

Example

```
$ awk  '{print "Marks of  "$1" are " $2,$3,$4}' Student
Marks of  Viji are 90 85 98
Marks of  Meera are 90 90 90
Marks of  Ann are 98 98 76
Marks of  Vasu are 89 99 91
Marks of  Sai are 98 87 97
Marks of  Hema are 98 87 76
```

BEGIN and END patterns are used to display header and footer information for an awk program. BEGIN pattern is used to initialize the variables, display header before processing the file. The END pattern is used to print information after processing the file. In the following example, BEGIN pattern is used to initialize the variable Total and print the header information "Name Marks"

```
BEGIN{Total=0; print "Name Marks"}
```

The body of the program processes each record in the file and calculates the total marks .

```
{Total=$2+$3+$4; print  $1 " " Total}
```

END pattern is used to display the Last Record number processed.

```
END{print "Total Students" NR}
```

NOTE

Information or variables have to enclosed within curly braces for BEGIN and END patterns. The body of the program also has to be enclosed within curly braces.

Example

```
$ awk  'BEGIN{Total=0; print "Name Marks"}{Total=$2+$3+$4; print  $1
" " Total}END{print "Total Students" NR}' Student
Name Marks
Viji 273
Meera 270
Ann 272
Vasu 279
Sai 282
Hema 261
Total Students6
```

'if' Statement

The structure of the if statement is

```
If(expression)
{
}
else if (expression)
{
}
else
{
}
```

The integrity of the expression is checked and if it is found to be true. The statements belonging to that if are executed. The following program calculates the Grade based on the Second mark of each student.

Example

```
$ awk   ' { if ( $2 > 97) {  print $1 " Grade A"} else if($2 > 93 &&
$2 < 96) {
>   print  $1 " Grade B"
> }
> else
> {
>   print $1" Grade C"
> }} ' Student

Viji Grade C
Meera Grade C
Ann Grade A
Vasu Grade C
Sai Grade A
Hema Grade A
```


While Loop

The structure of the 'while' loop is

```
While(expression)
{
       Statements…….
}
```

The integrity of the expression provided is checked on entry and as long as it is true, the loop is executed. The 'break' statement will take the program execution out of the loop and the 'continue' statement will take it back to the beginning of the loop.

Example

```
$ awk 'BEGIN{counter=0;print "Loop is starting";
for(counter=1;counter<5;counter=counter+1)
{
  print counter;
}}'
Loop is starting
1
2
3
4
```

For Loop structure

The for loop has two forms.The structure of the first form is composed of three parts. The first part of the loop does the initialization. Second part does the conditional checking and the third part does the manipulation of data.

```
for(initialization;expression;command/increment/decrement)
{
   statements…….
}
```

The structure for the second form is as follows

```
for(variable in array_number)
{
            Statements…….
}
```

Arrays will be discussed later in this session.

The following example is similar to while loop example that was explained earlier.

Example

```
$ awk 'BEGIN{counter=0;print "Loop is starting"
for(counter=1;counter<5;counter=counter+1)
{
  print counter;
}}'
Loop is starting
1
2
3
4
```

REVIEW QUESTIONS

Fill in the blanks

1. awk stands for_____
2. awk is used for _____,_____,_____,_____ and _____
3. The GNU implementation of awk is called _____
4. Variable names can consist of _____, _____ and _____.
5. To display all records which match the pattern starting with 'A', the command is_____
6. To display all records which match the pattern ending with 'n', the command is_____
7. To display Header and Footer information, _____ and _____ pattern can be used.

State true or false

1. To display the numbers from 1 to 10, the command is $awk '{x=0;while(x<10){print x;x=x+1}}' Student
2. The above command will display only once the range of numbers from 0 to 9.

Regular Expressions

A regular expression, or `regexp`, is a way of describing a set of strings. Because regular expressions are such a fundamental part of awk programming, their format and use deserve a say in this session

A regular expression enclosed in slashes (`/') is an awk pattern that matches every input record whose text belongs to that set.

The simplest regular expression is a sequence of letters, numbers, or both. Such a regexp matches any string that contains that sequence. Thus, the regexp `foo' matches any string containing `foo'.

Therefore, the pattern /foo/ matches any input record containing the three characters `foo', *anywhere* in the record. Other kinds of regexps let you specify more complicated classes of strings.

How to Use Regular Expressions

A regular expression can be used as a pattern by enclosing it in slashes. Then the regular expression is tested against the entire text of each record. (Normally, it only needs to match some part of the text in order to succeed.) For example, this prints the second field of each record that contains the three character `V' anywhere in it:

$ awk '/V/ { print $1 }' Student
Viji
Vasu

Regular expressions can also be used in matching expressions. These expressions allow you to specify the string to match against; it need not be the entire current input record. The two operators, `~' and `!~', perform regular expression comparisons. Expressions using these operators can be used as patterns or in if, while, for, and do statements.

exp ~ /regexp/

This is true if the expression *exp* (taken as a string) is matched by *regexp*. The following example matches, or selects, all input records with the upper-case letter 'V' somewhere in the first field:

```
$awk '$1 ~ /V/' Student
Viji   90 85 98 100
Vasu   89 99 91 87
```

exp *!*~ /regexp/

This is true if the expression *exp* (taken as a character string) is *not* matched by *regexp*. The following example matches, or selects, all input records whose first field *does not* contain the upper-case letter 'V':

```
$ awk '$1 !~ /V/' Student
Meera  90 90 90 100
Ann    98 98 76 97
Sai    98 87 97 96
Hema   98 87 76 97
```

When a regexp is written enclosed in slashes, like /foo/, we call it a *regexp constant*, much like 5.27 is a numeric constant, and "foo" is a string constant.

Escape Sequences

Some characters cannot be included literally in string constants ("foo") or regexp constants (/foo/). You represent them instead with *escape sequences*, which are character sequences beginning with a backslash ('\').

One use of an escape sequence is to include a double-quote character in a string constant. Since a plain double-quote would end the string, you must use `\"' to represent an actual double-quote character as a part of the string. For example:

$ awk 'BEGIN { print "He said \"hi!\" to her." }'

He said "hi!" to her.

The backslash character itself is another character that cannot be included normally; you write `\\' to put one backslash in the string or regexp.

Another use of backslash is to represent unprintable characters such as tab or newline. While there is nothing to stop you from entering most unprintable characters directly in a string constant or regexp constant, they may look ugly.

Here is a table of all the escape sequences used in awk, and what they represent. Unless noted otherwise, all of these escape sequences apply to both string constants and regexp constants.

\\

A literal backslash, `\'.

\a

The "alert" character, **Control-g**, ASCII code 7 (BEL).

\b

Backspace, **Control-h**, ASCII code 8 (BS).

\f

Formfeed, **Control-l**, ASCII code 12 (FF).

\n

Newline, **Control-j**, ASCII code 10 (LF).

\r

Carriage return, **Control-m**, ASCII code 13 (CR).

\t

Horizontal tab, **Control-i**, ASCII code 9 (HT).

\v

Vertical tab, **Control-k**, ASCII code 11 (VT).

nnn

The octal value *nnn*, where *nnn* are one to three digits between `0' and `7'. For example, the code for the ASCII ESC (escape) character is `\033'.

\x*hh*...

The hexadecimal value *hh*, where *hh* are hexadecimal digits (`0' through `9' and either `A' through `F' or `a' through `f'). Like the same construct in ANSI C, the escape sequence continues until the first non-hexadecimal digit is seen. However, using more than two hexadecimal digits produces undefined results. (The `\x' escape sequence is not allowed in POSIX awk.)

\/

A literal slash (necessary for regexp constants only). You use this when you wish to write a regexp constant that contains a slash. Since the regexp is delimited by slashes, you need to escape the slash that is part of the pattern, in order to tell awk to keep processing the rest of the regexp.

\"

A literal double-quote (necessary for string constants only). You use this when you wish to write a string constant that contains a double-quote. Since the string is delimited by double-quotes, you need to escape the quote that is part of the string, in order to tell awk to keep processing the rest of the string.

In a string constant, what happens if you place a backslash before something that is not one of the characters listed above? POSIX awk purposely leaves this case undefined. There are two choices.

- Strip the backslash out. This is what Unix awk and gawk both do. For example, "a\qc" is the same as "aqc".
- Leave the backslash alone. Some other awk implementations do this. In such implementations, "a\qc" is the same as if you had typed "a\\qc".

Regular Expression Operators

You can combine regular expressions with the following characters, called *regular expression operators*, or *metacharacters*, to increase the power and versatility of regular expressions.
The escape sequences described above are valid inside a regexp. They are introduced by a `\'. They are recognized and converted into the corresponding real characters as the very first step in processing regexps.

Here is a table of metacharacters. All characters that are not escape sequences and that are not listed in the table stand for themselves.

Meta character	**Explanation**	**Example**
`\`	This is used to suppress the special meaning of a character when matching.	`awk 'BEGIN { print "He said \"hi!\" to her \a." }'`
`^`	This matches the beginning of a string.	`awk '{ if ($1 ~ /^V/) print }' Student`
`$`	This is similar to `^', but it matches only at the end of a string.	`awk '{ if ($1 ~ /$a/) print }' Student`
`The period or dot`	This matches any single character, *including* the newline character.	`awk '{ if ($1 ~ /^.n/) print }' Student`
`[...]`	This is called a *character list*. It matches any *one* of the characters that are enclosed in the square brackets.	1. `awk '{ if ($1 ~ /^[AV]/) print }' Student` 2. `awk '{ if ($2 ~ /^[789]/) print $2 }' Student` 3. `awk '{ if ($2 ~ /^[7-9]/) print $2 }' Student`

Character classes are a new feature introduced in the POSIX standard. A character class is a special notation for describing lists of characters that have a specific attribute, but where the actual characters themselves can vary from country to country and/or from character set to character set. For example, the notion of what is an alphabetic character differs in the USA and in France.

A character class is only valid in a regexp *inside* the brackets of a character list. Character classes consist of `[:', a keyword denoting the class, and `:]'. Here are the character classes defined by the POSIX standard.

awk '{ if ($2 ~ /^[[:alpha:]]/) print $2 }' BBS-list
awk '{ if ($2 ~ /^[[:alnum:]]/) print $2 }' BBS-list

[:alnum:] → Alphanumeric characters.

[:alpha:] → Alphabetic characters.

[:blank:]→ Space and tab characters.

[:cntrl:] → Control characters.

[:digit:] → Numeric characters.

[:graph:] →Characters that are printable and are also visible. (A space is printable, but not visible, while an `a' is both.)

[:lower:] →Lower-case alphabetic characters.

[:print:] → Printable characters (characters that are not control characters.)

[:punct:] →Punctuation characters (characters that are not letter, digits, control characters, or space characters).

[:space:] →Space characters (such as space, tab, and formfeed, to name a few).

[:upper:] →Upper-case alphabetic characters.

[:xdigit:] →Characters that are hexadecimal digits.

For example, before the POSIX standard, to match alphanumeric characters, you had to write /[A-Za-z0-9]/. If your character set had other alphabetic characters in it, this would not match them. With the POSIX character classes, you can write /[[:alnum:]]/, and this will match *all* the alphabetic and numeric characters in your character set.

Two additional special sequences can appear in character lists. These apply to non-ASCII character sets, which can have single symbols (called *collating elements*) that are represented with more than one character, as well as several characters that are equivalent for *collating*, or sorting, purposes. (E.g., in French, a plain "e" and a grave-accented "`e" are equivalent.)

Collating Symbols

A *collating symbol* is a multi-character collating element enclosed in `[.' and `.]'. For example, if `ch' is a collating element, then [[.ch.]] is a regexp that matches this collating element, while [ch] is a regexp that matches either `c' or `h'.

Equivalence Classes

An *equivalence class* is a list of equivalent characters enclosed in `[=' and `=]'. Thus, [[=e`e=]] is regexp that matches either `e' or ``e'.

These features are very valuable in non-English speaking locales.

Arrays in 'Awk'

'awk' suports single dimensional arrays and since variables need not be defined prior to their being used. Arrays also need not be defined prior. The size of an array can dynamically grow. Arrays can be named by relating an index with a name. The index can be a number or a string.

`Arr[NR]=$0`—This statement stores the entire file into an array called Arr.
`Arr[0] or Arr["one"]` represent the first record

The program below displays the maximum mark got in the first field

Example

```
$awk    '{arr1[NR]=$2}END{max=arr1[0];
for(counter=0;counter<NR;counter=counter+1){
if(arr1[counter]>max) {max=arr1[counter]}}print "the maximum mark
is" max}' Student

the maximum mark is98
```

Functions in Awk

'awk' supports the following functions

- index(str1,str2)—returns the index of str1 in str2. Returns 0 if str2 is not available in str1.

Example

```
$awk '{ i=index($1,"Va"); print i }' Student
0
0
0
1
0
0
```

- length(str) ----returns the number of characters in the first field.

Example

```
$awk '{ i=length($1); print i }' Student
4
5
3
4
3
4
```

- substr(str,pos,n)---returns the substring of 'str' starting at pos with n characters

Example

```
$awk '{ i=substr($1,2,3); print i }' Student
iji
eer
nn
asu
ai
ema
```

Shell and Awk

The major disadvantage of 'Awk' is that it does not have an input command to read data from the keyboard. This is overcome by using 'awk' with the shell scripts.

Create a file called awkshell and add the following statement in the file and execute the shell script from the command line as given after the line.

Example

```
Echo "Enter the Student name"
read name
echo "The marks of the student $name are"
awk '{ if($1 == "'$name'") print $0}' Student
-----------------------------------------------------------------
-----------------------------------------------------
$sh awkshell
Enter the Student name
Viji

The marks of the student Viji are
Viji   90 85 98 100
```

Useful One Line Programs

Many useful awk programs are short, just a line or two. Here is a collection of useful, short programs to get you started. Most of the examples use a data file named `data'. This is just a placeholder; if you were to use these programs yourself, you would substitute your own file names for `data'.

awk '{ if (length($0) > max) max = length($0) }

END { print max }' data

This program prints the length of the longest input line.

awk 'length($0) > 80' data

This program prints every line that is longer than 80 characters. The sole rule has a relational expression as its pattern, and has no action (so the default action, printing the record, is used).

expand data | awk '{ if (x < length()) x = length() }

END { print "maximum line length is " x }'

This program prints the length of the longest line in `data'. The input is processed by the expand program to change tabs into spaces, so the widths compared are actually the right-margin columns.

awk 'NF > 0' data

This program prints every line that has at least one field. This is an easy way to delete blank lines from a file (or rather, to create a new file similar to the old file but from which the blank lines have been deleted).

awk 'BEGIN { for (i = 1; i <= 7; i++)

print int(101 * rand()) }'

This program prints seven random numbers from zero to 100, inclusive.

ls -lg * | awk '{ x += $5 } ; END { print "total bytes: " x }'

This program prints the total number of bytes used by *files*.

ls -lg * | awk '{ x += $5 }

END { print "total K-bytes: " (x + 1023)/1024 }'

This program prints the total number of kilobytes used by *files*.

awk -F: '{ print $1 }' /etc/passwd | sort

This program prints a sorted list of the login names of all users.

awk 'END { print NR }' data

This program counts lines in a file.

awk 'NR % 2' data

This program prints the even numbered lines in the data file. If you were to use the expression `NR % 2 == 1' instead, it would print the odd number lines.

REVIEW QUESTIONS

Fill in the blanks

1. _____ or _____ operator is used to represent equality
2. _____ is used to negate the special meaning of a character
3. To display the length of the entire record, the command is _____
4. To display the count of records, _____ Built in variable is used

State true or false

1. Awk cannot be used with the shell
2. Awk can accept input.
3. Awk supports multi dimensional arrays

☞TOTAL RECALL

1. What is the purpose of awk programming?
2. How can expression be specified in awk?
3. When a shell variable is used in awk, How should it be specified?
4. What is the syntax for for loop?
5. What are the built in variables in awk?