AWK Tutorial

AWK is a tool designed for text processing. It is often used for data munging (pre-processing, formatting, extraction, etc.). AWK was developed in 1977 by Alfred **A**ho, Peter **W**einberger, and Brian **K**ernighan. The name AWK comes from the last initials of the three developers.

In this tutorial, we will introduce you to some of the basic and most often used features of AWK. The beauty of AWK is that it is easy to learn and can be used to accomplish numerous tasks without the need to write long programs.

Basic Structure

AWK scripts process text files one line at a time (i.e. it is line oriented). It has the following implicit loop:

for each line in input do
    execute the commands in the AWK script.

An AWK script has the following form:

```
pattern { action }
```

pattern specifies when the `action` is performed. For example, two special patterns are **BEGIN** and **END**. The BEGIN pattern specifies a block of actions that should be performed once, at the beginning of processing. Similarly, the END pattern specifies a block of actions to be done once, at the end of processing. A **null pattern** (a blank pattern) specifies that its action block should be executed on each line of input. Here is a simple example script (stored in a file: **`helloAWK.awk`**):

```
BEGIN { print “Script Started” }
{ print $0 }                       # A line with a null pattern
END { print “Script Completed.” }
```

The script above specifies the printing of two start and end messages. A **#** represents the start of a comment. The second line (a null pattern) prints out the contents of each line. **`$0`** is an AWK variable (more on that below) that gets bound to each line. For example, suppose the input file (**trigram.txt**) has the following contents:

```
He likes attention 196
He likes bananas 51
He likes barbeque 44
He likes baseball 188
He likes basketball 57
He likes beer 281
He likes being 2026
```

The above data is from Google’s trigram dataset. Each line represents a sequence of three words followed by a count of how many times the sequence of three words appeared on the web. To run the above script on the data file, we use the command:

```
$ awk -f hello.awk trigram.txt
Script Started
He likes attention 196
He likes bananas 51
He likes barbeque 44
He likes baseball 188
He likes basketball 57
He likes beer 281
He likes being 2026
Script Completed.
```

The **-f** option on the awk command is followed by the name of the script file (`helloAWK.awk`) which is followed by the name of the text file (`trigram.txt`). We can also run the command using I/O redirection:

```
$ awk -f hello.awk < trigram.txt
```

In fact, for short AWK scripts we can even specify the script in the command line:

```
$ awk ‘{print $0}’ trigram.txt
He likes attention 196
He likes bananas 51
He likes barbeque 44
He likes baseball 188
He likes basketball 57
He likes beer 281
He likes being 2026
```

AWK Variables

AWK defines several variables that are helpful in writing scripts and processing the input line. In the `helloAWK.awk` script, we used a variable **`$0`**. For each line, AWK binds `$0` to the contents of the entire input line. In addition, AWK treats Each item on a line as a *field*. It creates and binds variables to each field. Thus, for the first line of the text file we will have the following variables and bindings:

```
$0: “He likes attention 196”
$1: “He”
$2: “likes”
$3: “attention”
$4: 196
```

A field on a line is anything separated by a space character. Suppose we wanted to process the input file so that only the third word and the counts were output, we could write the following short script:

```
{ print $3, $4 }
```

Let us run and observe the output:

```
$ awk ‘{print $3, $4}’ trigram.txt
attention 196
bananas 51
barbeque 44
baseball 188
basketball 57
beer 281
being 2026
```

We can also switch the order. Say, we want to see the counts to appear at the start of each line:

```
$ awk ‘{print $4, $1, $2, $3}’ trigram.txt
196 He likes attention
51 He likes bananas
44 He likes barbeque
188 He likes baseball
57 He likes basketball
281 He likes beer 281
2026 He likes being 2026
```

As you can see, AWK can be used as a valuable tool to do data munging (processing, reformatting, etc.) to prepare for an application. For example, say we want to prepare the file so it can be a CSV file (comma separated file):

```
$ awk ‘{print $1,”,”,$2,”,”,$3,”,”,$4}’ trigram.txt
He ,likes ,attention ,196
He ,likes ,bananas ,51
He ,likes ,barbeque ,44
He ,likes ,baseball ,188
He ,likes ,basketball ,57
He ,likes ,beer ,281
He ,likes ,being ,2026
```

Well, that inserted the commas, but didn’t quite come out right. That is because the print command always appends a space character after each output. AWK also accepts the C **`printf()`** command. This allows us to have more control over formatting:

```
$ awk ‘printf(“%s,%s,%s,%d\n”, $1, $2, $3, $4)}’ trigram.txt
He,likes,attention,196
He,likes,bananas,51
He,likes,barbeque,44
He,likes,baseball,188
He,likes,basketball,57
He,likes,beer,281
He,likes,being,2026
```

We mentioned above that the fields in a line need to be separated by a space character. We can specify a field separator in the script as well. For example, consider a CSV file with these contents (`trigram2.txt`):

```
He,likes,attention,196
He,likes,bananas,51
He,likes,barbeque,44
He,likes,baseball,188
He,likes,basketball,57
He,likes,beer,281
He,likes,being,2026
```

We can use the command line option **-F** to specify that the field separator is a comma (**,**) instead of a space:

```
$ awk -F, '{print $4, $1, $2, $3}' trigram2.txt
196 He likes attention
51 He likes bananas
44 He likes barbeque
188 He likes baseball
57 He likes basketball
281 He likes beer 281
2026 He likes being 2026
```

In fact, AWK predefines an input field separator variable, FS that can also be set at the beginning of a script (say, in file helloAWK2.awk):

```
# File: helloAWK2.awk
BEGIN { FS=","; }
{ print $4, $1, $2, $3, $4 }
```

```
$ awk -f helloAWK2.awk trigram2.txt
196 He likes attention
51 He likes bananas
44 He likes barbeque
188 He likes baseball
57 He likes basketball
281 He likes beer 281
2026 He likes being 2026
```

Other AWK Variables

In addition to the $-variables, that are automatically created for each line, and FS, AWK defined several other useful variables:

FS	Field Separator
NF	Number of fields on the current line
NR	Record/line number
FILENAME	Name of the current file being processed

These variables can be referenced in an awk script by their names (listed above). For example,

```
$ awk '{print NR, $0}' trigram.txt
1 He likes attention 196
2 He likes bananas 51
3 He likes barbeque 44
4 He likes baseball 188
5 He likes basketball 57
6 He likes beer 281
7 He likes being 2026
```

You can also use the named variable with a **$**-prefix to access a field in a line. For example, to extract the value of the last field from a data file:

```
$ awk '{print $NR}' trigram.txt
196
51
44
188
57
281
2026
```

User Defined Variables

At any time in a script a user-named variable can be defined and used (file `total.awk`):

```
BEGIN { sum=0; }
{ sum = sum + $4;
  print $0;
}
END { printf("The total count is %d\n", sum); }
```

```
$ awk -f total.awk trigram.txt
He likes attention 196
He likes bananas 51
He likes barbeque 44
He likes baseball 188
He likes basketball 57
He likes beer 281
He likes being 2026
The total count is 2843
```

Example: Normalizing counts

Say we want to normalize the counts in our data file. That is, divide each count by the total. To do this, first we will compute the count, as above. Next, we can write the following script (`normalize.awk`):

```
BEGIN { total=2843; }
{ norm = $4/total;
  printf("%s %s %s %0.3f\n", $1, $2, $3, norm);
}
```

```
$ awk -f normalize.awk trigram.txt
He likes attention 0.069
He likes bananas 0.018
He likes barbeque 0.015
He likes baseball 0.066
He likes basketball 0.020
He likes beer 0.099
He likes being 0.713
```

Executable AWK Scripts

Any AWK script file can be turned into an executable shell script. This can be done in two steps. First, add the script line to the AWK source file (we will use `total.awk` from above):

```
#!/usr/bin/awk -f
BEGIN { sum=0; }
{ sum = sum + $4;
  print $0;
}
END { printf("The total count is %d\n", sum); }
```

Next, make the script file have executable permissions:

```
$ chmod 755 total.awk
```

Now the script can be executed like a Linux command:

```
$ ./total.awk trigram.txt
He likes attention 196
He likes bananas 51
He likes barbeque 44
He likes baseball 188
He likes basketball 57
He likes beer 281
He likes being 2026
The total count is 2843
```

AWK Statements

AWK has all C-like statements available to use in the scripts. That is, it is a general-purpose programming language. Here is the list of all the statements available:

```
if ( conditional ) statement [ else statement ]
while ( conditional ) statement
for ( expression ; conditional ; expression ) statement
for ( variable in array ) statement
break
continue
{ [ statement ] ...}
variable=expression
```

```
print [ expression-list ] [ > expression ]
printf format [ , expression-list ] [ > expression ]
next
exit
```

The **next** command, if executed, moves on to processing the next line in input. And the **exit** command quits the script.

Example: Filter out any lines in the input with counts less than 60. Here is the script file (`filter.awk`):

```
#!/usr/bin/awk -f
BEGIN { thresh=60; }
{ if ($4 <= 60)
      next;       # Skip this line
   else
      print $1, $2, $3, $4;
}
```

```
$ ./filter.awk trigram.txt
He likes attention 196
He likes baseball 188
He likes beer 281
He likes being 2026
```

AWK Patterns

At the start of this tutorial, we mentioned that an AWK script has the following syntax:

```
pattern { action }
```

So far, we have only been using the BEGIN, END, and the null patterns. A pattern essentially specifies a condition (a test). If the pattern matches the input line (or the test succeeds) the associated block's action is carried out. For example, look at the script, `filter.awk`:

```
#!/usr/bin/awk -f
BEGIN { thresh=60; }
{ if ($4 <= 60)
      next;
   else
      print $1, $2, $3, $4;
}
```

Using a pattern, we could write the script where the condition is specified as a pattern:

```
#!/usr/bin/awk -f
BEGIN { thresh=60; }
$4 <= 60 { next; }
$4 > 60 { print $1, $2, $3, $4;}
```

Another way to specify patterns is as regular expressions (remember when we learned the **`grep`** commands?). Regular Expressions are a convenient way to define patterns in strings. Thus, the regular expression pattern **`/likes/`** will match any line that contains the pattern. For example: The script (`pattern.awk`):

```
/ba/ { print $0; }
```

will print all lines in the input that have the pattern "**ba**" in them. Let's run the script:

```
$ awk -f pattern.awk trigram
He likes bananas 51
He likes barbeque 44
He likes baseball 188
He likes basketball 57
```

Above, any line that matches the pattern is printed (the action). If we wanted to eliminate all lines that contain the pattern **`/ba/`** we could write the following:

```
/ba/ { next; }    # Skip this line as it has /ba/
{ print $0;}      # Otherwise print it
```

```
$ awk -f pattern.awk trigram
He likes attention 196
He likes beer 281
He likes being 2026
```

But wait! There is more!!

AWK has several library functions that can be used:

Trigonomentric Functions	`sin()`, `cos()`, etc.
Math Functions	`exp()`, `log()`, `sqrt()`, etc.
Random Numbers	`srand()`, `rand()`
String Functions	`length()`, `split()`, `tolower()`, `toupper()`, etc.
User-defined Functions	Yes, this is also possible.

This is a good place to stop. As you can see, AWK is a useful tool to do any kind of text/data file munging. We have introduced some of the key features of AWK in this tutorial. Perhaps the most often used features you will ever need. Consult the AWK Manual for more. During the break, learn about Regular Expressions. They are very useful in several contexts.