Nov 29, 2022 • 21 min read

Getting Started With AWK Command [Beginner's Guide]

Sylvain Leroux

Table of Contents

The AWK command dates back to the early Unix days. It is part of the POSIX standard and should be available on any Unix-like system. And beyond.

While sometimes discredited because of its age or lack of features compared to a multipurpose language like Perl, AWK remains a tool I like to use in my everyday work. Sometimes for writing relatively complex programs, but also because of the powerful one-liners you can write to solve issues with your data files.

So, this is exactly the purpose of this article. Showing you how you can leverage the AWK power in less than 80 characters to perform useful tasks. This article is not intended to be a complete AWK tutorial, but I have still included some basic commands at the start so even if you have little to no previous experience you can grab the core AWK concepts.

My sample files for this AWK tutorial

All the one-liners described in that article will be tested on the same data file:

```
cat file
CREDITS,EXPDATE,USER,GROUPS
99,01 jun 2018,sylvain,team:::admin
52,01    dec   2018,sonia,team
52,01    dec   2018,sonia,team
25,01    jan   2019,sonia,team
10,01 jan 2019,sylvain,team:::admin
8,12    jun   2018,öle,team:support



17,05 apr 2019,abhishek,guest
```

You can get a copy of that file online at GitHub.

Know Pre-defined and automatic variables in AWK

AWK supports a couple of pre-defined and automatic *variables* to help you write your programs. Among them you will often encounter:

RS –*The record separator.* AWK processes your data one record at a time. The record separator is the delimiter used to split the input data stream into records. By default, this is the newline character. So if you do not change it, a record is one line of the input file.

NR – *The current input record number.* If you are using the standard newline delimiter for your records, this match with the current input line number.

FS/OFS –*The character(s) used as the field separator.* Once AWK reads a record, it splits it into different fields based on the value of `FS`. When AWK print a record on the output, it will rejoin the fields, but this time, using the `OFS` separator instead of the `FS` separator. Usually, `FS` and `OFS` are the same, but this is not mandatory. “white space” is the default value for both of them.

NF – The number of fields in the current record. If you are using the standard "white space" delimiter for your fields, this will match with the number of words in the current record.

There are other more or less standard AWK variables available, so it worth checking your particular AWK implementation manual for more details. However, this subset is already enough to start writing interesting one-liners.

A. Basic usage of AWK command

1. Print all lines

This example is mostly useless, but it will nevertheless be a good introduction the AWK syntax:

```
awk '1 { print }' file
CREDITS,EXPDATE,USER,GROUPS
99,01 jun 2018,sylvain,team:::admin
52,01    dec   2018,sonia,team
52,01    dec   2018,sonia,team
25,01    jan   2019,sonia,team
10,01 jan 2019,sylvain,team:::admin
8,12    jun   2018,öle,team:support



17,05 apr 2019,abhishek,guest
```

AWK programs are made of one or many `pattern { action }` statements.

If, for a given *record* ("line") of the input file, the *pattern* evaluates to a non-zero value (equivalent to "true" in AWK), the commands in the corresponding *action block* are executed. In the above example, since `1` is a non-zero constant, the `{ print }` action block is executed for each input record.

Another trick is `{ print }` is the default action block that will be used by AWK if you do not explicitly specify one. So the above command can be shortened as:

```
awk 1 file
CREDITS,EXPDATE,USER,GROUPS
99,01 jun 2018,sylvain,team:::admin
52,01    dec   2018,sonia,team
52,01    dec   2018,sonia,team
25,01    jan   2019,sonia,team
10,01 jan 2019,sylvain,team:::admin
8,12    jun   2018,öle,team:support



17,05 apr 2019,abhishek,guest
```

Almost as useless, the following AWK program will consume its input but will not produce anything to the output:

```
awk 0 file
```

2. Remove a file header

```
awk 'NR>1' file
99,01 jun 2018,sylvain,team:::admin
52,01    dec   2018,sonia,team
52,01    dec   2018,sonia,team
25,01    jan   2019,sonia,team
10,01 jan 2019,sylvain,team:::admin
8,12    jun   2018,öle,team:support



17,05 apr 2019,abhishek,guest
```

Remember, this is the equivalent of writing explicitly:

```
awk 'NR>1 { print }' file
99,01 jun 2018,sylvain,team:::admin
52,01    dec   2018,sonia,team
52,01    dec   2018,sonia,team
25,01    jan   2019,sonia,team
10,01 jan 2019,sylvain,team:::admin
8,12    jun   2018,öle,team:support



17,05 apr 2019,abhishek,guest
```

This one-liner will write records of the input file except for the very first one since in that case the condition is `1>1` which obviously is not true.

Since this program is using the default values for `RS`, in practice it will discard the first line of the input file.

3. Print lines in a range

This is just a generalization of the preceding example, and it does not deserve many explanations, except to say `&&` is the logical `and` operator:

```
awk 'NR>1 && NR < 4' file
99,01 jun 2018,sylvain,team:::admin
52,01    dec   2018,sonia,team
```

4. Removing whitespace-only lines

```
awk 'NF' file
CREDITS,EXPDATE,USER,GROUPS
99,01 jun 2018,sylvain,team:::admin
52,01    dec   2018,sonia,team
52,01    dec   2018,sonia,team
25,01    jan   2019,sonia,team
10,01 jan 2019,sylvain,team:::admin
8,12    jun   2018,öle,team:support
17,05 apr 2019,abhishek,guest
```

AWK splits each record into fields, based on the field separator specified in the `FS` variable. The default field separator is *one-or-several-white-space-characters* (aka, spaces or tabs). With those settings, any record containing at least one non-whitespace character will contain at least one field.

In other words, the only case where `NF` is 0 ("false") is when the record contains only spaces. So, that one-liner will only print records containing at least one non-space character.

5. Removing all blank lines

```
awk '1' RS='' file
CREDITS,EXPDATE,USER,GROUPS
99,01 jun 2018,sylvain,team:::admin
52,01    dec   2018,sonia,team
52,01    dec   2018,sonia,team
25,01    jan   2019,sonia,team
10,01 jan 2019,sylvain,team:::admin
8,12    jun   2018,öle,team:support

17,05 apr 2019,abhishek,guest
```

This one-liner is based on an obscure POSIX rule that specifies if the `RS` is set to the empty string, *"then records are separated by sequences consisting of a <newline> plus one or more blank lines."*

Worth mentioning in the POSIX terminology, a blank line is a completely empty line. Lines that contain only whitespace do not count as "blank".

6. Extracting fields

This is probably one of the most common use cases for AWK: extracting some columns of the data file.

```
awk '{ print $1, $3}' FS=, OFS=, file
CREDITS,USER
99,sylvain
52,sonia
52,sonia
25,sonia
10,sylvain
8,öle
        ,
,
,
17,abhishek
```

Here, I explicitly set both the input and output field separators to the coma. When AWK split a record into fields, it stores the content of the first field into $1, the content of the second field into $2 and so on. I do not use that here, but worth mentioning $0 is the entire record.

In this one-liner, you may have noticed I use an action block without a pattern. In that case, 1 ("true") is assumed for the pattern, so the action block is executed for each record.

Depending on your needs, it may not produce what we would like for blank or whitespace-only lines. In that case, that second version could be a little bit better:

```
awk 'NF { print $1, $3 }' FS=, OFS=, file
CREDITS,USER
99,sylvain
52,sonia
52,sonia
25,sonia
10,sylvain
8,öle
        ,
17,abhishek
```

In both cases, I passed custom values for `FS` and `OFS` on the command line. Another option would be to use a special `BEGIN` block inside the AWK program to initialize those variables before the first record is read. So, depending on your taste, you may prefer writing that instead:

```
awk 'BEGIN { FS=OFS="," } NF { print $1, $3 }' file
CREDITS,USER
99,sylvain
52,sonia
52,sonia
25,sonia
10,sylvain
8,öle
        ,
17,abhishek
```

Worth mentioning here you may also use `END` blocks to perform some tasks after the last record has been read. Like we will see it just now. That being said, I admit this is far from being perfect since whitespace-only lines are not handled elegantly. We will soon see a possible solution, but before that let's do some math...

7. Performing calculations column-wise

AWK support the standard arithmetical operators. And will convert values between text and numbers automatically depending on the context. Also, you can use your own variables to store intermediate values. All that allows you to write compact programs to perform calculations on data columns:

```
awk '{ SUM=SUM+$1 } END { print SUM }' FS=, OFS=, file
263
```

Or, equivalently using the `+=` shorthand syntax:

```
awk '{ SUM+=$1 } END { print SUM }' FS=, OFS=, file
263
```

Please note AWK variables do not need to be declared before usage. An undefined variable is assumed to hold the empty string. Which, according to the AWK type conversion rules, is equal the 0 number. Because of that feature, I did not bother

handling explicitly the case where `$1` contains text (in the heading), whitespace or just nothing. In all those cases, it will count as 0 and will not interfere with our summation. Of course, it would be different if I performed multiplications instead. So, why wouldn't you use the comment section to suggest a solution for that case?

8. Counting the number of non-empty lines

I have already mentioned the `END` rule before. Here is another possible application to count the number of non-empty lines in a file:

```
awk '/./ { COUNT+=1 } END { print COUNT }' file
9
```

Here I used the `COUNT` variable and incremented it (`+=1`) for each line matching the regular expression `/./`. That is each line containing at least one character. Finally, the END block is used to display the final result once the entire file has been processed. There is nothing special about the name `COUNT`. I could have used `Count`, `count`, `n`, `xxxx` or any other name complying with the AWK variable naming rules

However, is this result correct? Well, it depends on your definition of an "empty" line. If you consider only blank lines (according to POSIX) are empty, then this is correct. But maybe would you prefer considering whitespace-only lines as empty too?

```
awk 'NF { COUNT+=1 } END { print COUNT }' file
8
```

This time the result is different since that later version ignores whitespace-only lines too, whereas the initial version only ignored blank lines. Can you see the difference? I let you figure that by yourself. Don't hesitate to use the comment section if this isn't clear enough!

Finally, if you are interested only in data lines, and given my particular input data file, I could write that instead:

```
awk '+$1 { COUNT+=1 } END { print COUNT }' file
7
```

It works because of the AWK type conversion rules. The unary plus in the pattern forces the evaluation of $1 in a numerical context. In my file, data records contain a number in their first field. Non-data records (heading, blank lines, whitespace-only lines) contain text or nothing. All of them being equal to 0 when converted to numbers.

Notice with that latest solution, a record for a user eventually having 0 credits would be discarded too.

B. Using Arrays in AWK

Arrays are a powerful feature of AWK. All arrays in AWK are associative arrays, so they allow associating an arbitrary string with another value. If you are familiar with

other programming languages, you may know them as *hashes*, *associative tables*, *dictionaries* or *maps*.

9. A simple example of AWK array

Let's imagine I want to know the total credit for all users. I can store an entry for each user in an associative array, and each time I encounter a record for that user, I increment the corresponding value stored in the array.

```
awk '+$1 { CREDITS[$3]+=$1 }
     END { for (NAME in CREDITS) print NAME, CREDITS[NAME] }' FS=, file
abhishek 17
sonia 129
öle 8
sylvain 109
```

I admit this is no longer a one-liner. Mostly because of the `for` loop used to display the content of the array after the file has been processed. So, let's go back now to shorter examples:

10. Identifying duplicate lines using AWK

Arrays, just like other AWK variables, can be used both in action blocks as well as in patterns. By taking benefit of that, we can write a one-liner to print only duplicate lines:

```
awk 'a[$0]++' file
52,01    dec   2018,sonia,team
```

The `++` operator is the post-increment operator inherited from the C language family (whose AWK is a proud member, thanks to Brian Kernighan been one of its original authors).

As its name implies the post-increment operator increments ("add 1") a variable, but only after its value has been taken for the evaluation of the englobing expression.

In that case, `a[$0]` is evaluated to see if the record will be printed or not, and once the decision has been made, in all cases, the array entry is incremented.

So the first time a record is read, `a[$0]` is undefined, and thus equivalent to zero for AWK. So that first record is not written on the output. Then that entry is changed from zero to one.
The second time the same input record is read, `a[$0]` is now 1. That is "true". The line will be printed. However, before that, the array entry is updated from 1 to 2. And so on.

11. Removing duplicate lines

As a corollary of the previous one-liner, we may want to remove duplicate lines:

```
awk '!a[$0]++' file
CREDITS,EXPDATE,USER,GROUPS
99,01 jun 2018,sylvain,team:::admin
```

```
52,01    dec   2018,sonia,team
25,01    jan   2019,sonia,team
10,01 jan 2019,sylvain,team:::admin
8,12    jun   2018,öle,team:support


17,05 apr 2019,abhishek,guest
```

The only difference is the use of the logical, not operator (`!`) that reverse the truth value of the expression. What was false becomes true, and what was true becomes false. The logical not have absolutely no influence on the `++` post increment which works exactly as before.

C. Field and record separator magic

12. Changing the field separators

```
awk '$1=$1' FS=, OFS=';' file
CREDITS;EXPDATE;USER;GROUPS
99;01 jun 2018;sylvain;team:::admin
52;01    dec   2018;sonia;team
52;01    dec   2018;sonia;team
25;01    jan   2019;sonia;team
10;01 jan 2019;sylvain;team:::admin
8;12    jun   2018;öle;team:support

17;05 apr 2019;abhishek;guest
```

That program sets the `FS` and `OFS` variable to use a coma as input field separator and a semicolon as the output field separator. Since AWK does not change the output record as long as you did not change a field, the `$1=$1` trick is used to force AWK to break the record and reassemble it using the output field separator.

Remember here the default action block is `{ print }`. So you could rewrite that more explicitly as:

```
awk '$1=$1 { print }' FS=, OFS=';' file
CREDITS;EXPDATE;USER;GROUPS
99;01 jun 2018;sylvain;team:::admin
52;01    dec   2018;sonia;team
52;01    dec   2018;sonia;team
25;01    jan   2019;sonia;team
10;01 jan 2019;sylvain;team:::admin
8;12    jun   2018;öle;team:support

17;05 apr 2019;abhishek;guest
```

You may have noticed both those examples are removing empty lines too. Why? Well, remember the AWK conversion rules: an empty string is “false.” All other strings are “true.” The expression `$1=$1` is an affectation that alters `$1` . However, this is an expression too. And it evaluates to the value of `$1` –which is “false” for the empty string. If you really want all lines, you may need to write something like that instead:

```
awk '($1=$1) || 1 { print }' FS=, OFS=';' file
CREDITS;EXPDATE;USER;GROUPS
```

```
99;01 jun 2018;sylvain;team:::admin
52;01    dec   2018;sonia;team
52;01    dec   2018;sonia;team
25;01    jan   2019;sonia;team
10;01 jan 2019;sylvain;team:::admin
8;12    jun   2018;öle;team:support



17;05 apr 2019;abhishek;guest
```

Do you remember the `&&` operator? It was the logical AND. `||` is the logical OR. The parenthesis is necessary here because of the operators precedence rules. Without them, the pattern would have been erroneously interpreted as `$1=($1 || 1)` instead. I let as an exercise for you to test how the result would have been different then.

Finally, if you are not too keen about arithmetic, I bet you will prefer that simpler solution:

```
awk '{ $1=$1; print }' FS=, OFS=';' file
CREDITS;EXPDATE;USER;GROUPS
99;01 jun 2018;sylvain;team:::admin
52;01    dec   2018;sonia;team
52;01    dec   2018;sonia;team
25;01    jan   2019;sonia;team
10;01 jan 2019;sylvain;team:::admin
8;12    jun   2018;öle;team:support



17;05 apr 2019;abhishek;guest
```

13. Removing multiple spaces

```
awk '$1=$1' file
CREDITS,EXPDATE,USER,GROUPS
99,01 jun 2018,sylvain,team:::admin
52,01 dec 2018,sonia,team
52,01 dec 2018,sonia,team
25,01 jan 2019,sonia,team
10,01 jan 2019,sylvain,team:::admin
8,12 jun 2018,öle,team:support
17,05 apr 2019,abhishek,guest
```

This is almost the same program as the preceding one. However, I left the field separators to their default values. So, multiple whitespaces are used as the input field separator, but only one space is used as the output field separator. This has the nice side effect of coalescing *multiples* whitespaces into *one* space.

14. Joining lines using AWK

We have already used `OFS`, the output field separator. As you may have guessed, it has the `ORS` counterpart to specify the output record separator:

```
awk '{ print $3 }' FS=, ORS=' ' file; echo
USER sylvain sonia sonia sonia sylvain öle    abhishek
```

Here, I used a space after each record instead of a newline character. This one-liner is sufficient in some use cases, but it still has some drawbacks.

Most obviously, it does not discard whitespace-only lines (the extra spaces after *öle* are coming from that). So, I may end up using a plain regular expression instead:

```
awk '/[^[:space:]]/ { print $3 }' FS=, ORS=' ' file; echo
USER sylvain sonia sonia sonia sylvain öle abhishek
```

It is better now, but there is still a possible issue. It will be more obvious if we change the separator to something visible:

```
awk '/[^[:space:]]/ { print $3 }' FS=, ORS='+' file; echo
USER+sylvain+sonia+sonia+sonia+sylvain+öle+abhishek+
```

There is an extra separator at the end of the line— because the field separator is written *after* each record. Including the last one.

To fix that, I will rewrite the program to display a custom separator *before* the record, starting from the second output record.

```
awk '/[^[:space:]]/ { print SEP $3; SEP="+" }' FS=, ORS='' file; echo
USER+sylvain+sonia+sonia+sonia+sylvain+öle+abhishek
```

Since I take care of adding the separator by myself, I also set the standard AWK output record separator to the empty string. However, when you start dealing yourself with separators or formatting, it may be the sign you should consider using the printf function instead of the `print` statement. As we will see it right now.

D. Field formatting

I have already mentioned the relationship between the AWK and C programming languages. Among other things, from the C language standard library AWK inherits the powerful `printf` function, allowing great control over the formatting of the text sent to the output.

The `printf` function takes a format as the first argument, containing both plain text that will be output verbatim and wildcards used to format different section of the output. The wildcards are identified by the `%` character. The most common being `%s` (for string formatting), `%d` (for integer numbers formatting) and `%f` (for floating point number formatting). As this can be rather abstract, let's see an example:

```
awk '+$1 { printf("%s ",  $3) }' FS=, file; echo
sylvain sonia sonia sonia sylvain öle abhishek
```

You may notice, as the opposite of the `print` statement, the `printf` function does not use the `OFS` and `ORS` values. So, if you want some separator, you have to

explicitly mention it as I did by adding a space character at the end of the format string. This is the price to pay for having full control of the output.

While not at all a format specifier, this is an excellent occasion to introduce the `\n` notation which can be used in any AWK string to represent a newline character.

```
awk '+$1 { printf("%s\n",  $3) }' FS=, file
sylvain
sonia
sonia
sonia
sylvain
öle
abhishek
```

15. Producing tabular results

AWK enforces a record/field data format based on delimiters. However, using the `printf` function, you can also produce fixed-width tabular output. Because each format specifier in a `printf` statement can accept an optional width parameter:

```
awk '+$1 { printf("%10s | %4d\n",  $3, $1) }' FS=, file
   sylvain |   99
     sonia |   52
     sonia |   52
     sonia |   25
   sylvain |   10
       öle |    8
  abhishek |   17
```

As you can see, by specifying the width of each field, AWK pads them to the left with spaces. For text, it is usually preferable to pad on the right, something that can be achieved using a negative width number. Also, for integers, we may like to pad fields with zeros instead of spaces. This can be obtained by using an explicit 0 before the field width:

```
awk '+$1 { printf("%-10s | %04d\n",  $3, $1) }' FS=, file
sylvain    | 0099
sonia      | 0052
sonia      | 0052
sonia      | 0025
sylvain    | 0010
öle        | 0008
abhishek   | 0017
```

16. Dealing with floating point numbers

The `%f` format does not deserve much explanations...

```
awk '+$1 { SUM+=$1; NUM+=1 } END { printf("AVG=%f",SUM/NUM); }' FS=, fil
AVG=37.571429
```

... except maybe to say you almost always want to explicitly set the field width and precision of the displayed result:

```
awk '+$1 { SUM+=$1; NUM+=1 } END { printf("AVG=%6.1f",SUM/NUM); }' FS=,
AVG=  37.6
```

Here, the field width is 6, which means the field will occupy the space of 6 characters (including the dot, and eventually padded with spaces on the left as usually). The .1 precision means we want to display the number with 1 decimal numbers after the dot. I let you guess what `%06.1` would display instead.

E. Using string functions in AWK

In addition to the `printf` function, AWK contains few other nice string manipulation functions. In that domain, modern implementations like Gawk have a richer set of internal functions at the price of lower portability. As of myself, I will stick here with just a few POSIX-defined function that should work the same anywhere.

17. Converting text to upper case

This one, I use it a lot, because it handles internationalization issues nicely:

```
awk '$3 { print toupper($0); }' file
99,01 JUN 2018,SYLVAIN,TEAM:::ADMIN
52,01    DEC   2018,SONIA,TEAM
52,01    DEC   2018,SONIA,TEAM
25,01    JAN   2019,SONIA,TEAM
10,01 JAN 2019,SYLVAIN,TEAM:::ADMIN
8,12    JUN   2018,ÖLE,TEAM:SUPPORT
17,05 APR 2019,ABHISHEK,GUEST
```

As a matter of fact, this is probably the best and most portable solution to convert text to uppercase from the shell.

18. Changing part of a string

Using the `substr` command, you can split a string of characters at a given length. Here I use it to capitalize only the first character of the third field:

```
awk '{ $3 = toupper(substr($3,1,1)) substr($3,2) } $3' FS=, OFS=, file
CREDITS,EXPDATE,USER,GROUPS
99,01 jun 2018,Sylvain,team:::admin
52,01    dec   2018,Sonia,team
52,01    dec   2018,Sonia,team
25,01    jan   2019,Sonia,team
10,01 jan 2019,Sylvain,team:::admin
8,12    jun   2018,Öle,team:support
17,05 apr 2019,Abhishek,guest
```

The `substr` function takes the initial string, the (1-based) index of the first character to extract and the number of characters to extract. If that last argument is missing, `substr` takes all the remaining characters of the string.

So, `substr($3,1,1)` will evaluate to the first character of `$3`, and `substr($3,2)` to the remaining ones.

19. Splitting fields in sub-fields

The AWK record-field data model is really nice. However, sometimes you want to split fields themselves into several parts based on some internal separator:

```
awk '+$1 { split($2, DATE, " "); print $1,$3, DATE[2], DATE[3] }' FS=, O
99,sylvain,jun,2018
52,sonia,dec,2018
52,sonia,dec,2018
25,sonia,jan,2019
10,sylvain,jan,2019
8,öle,jun,2018
17,abhishek,apr,2019
```

Somewhat surprisingly, this works even if some of my fields are separated by more than one whitespace. Mostly for historical reasons, when the separator is a single space, `split` will consider "the elements are separated by runs of whitespace." And not only by just one. The `FS` special variable follows the same convention.

However, in the general case, one character string match one character. So, if you need something more complex, you have to remember the field separator is an extended regular expression.

As an example, let's see how would be handled the group field which appears to be a multivalued field using a colon as separator:

```
awk '+$1 { split($4, GRP, ":"); print $3, GRP[1], GRP[2] }' FS=, file
sylvain team
sonia team
sonia team
sonia team
sylvain team
öle team support
abhishek guest
```

Whereas I would have expected to display up to two groups per user, it shows only one for most of them. That issue is caused by the multiple occurrences of the separator. So, the solution is:

```
awk '+$1 { split($4, GRP, /:+/); print $3, GRP[1], GRP[2] }' FS=, file
sylvain team admin
sonia team
sonia team
sonia team
sylvain team admin
öle team support
abhishek guest
```

The slashes instead of the quotes denote the literal as being a regular expression rather than a plain string, and the plus sign indicates this expression will match one

or several occurrences of the previous character. So, in that case, each separator is made (of the longest sequence of) one or several consecutive colons.

How to Split String into Array in Bash [Easiest Way]

In this quick tip, you'll learn to split a string into an array in Bash script.

Linux Handbook • • Abhishek Prakash

20. Searching and replacing with AWK commands

Speaking of regular expressions, sometimes you want to perform substitution like the sed `s///g` command, but only on one field. The `gsub` command is what you need in that case:

```
awk '+$1 { gsub(/ +/, "-", $2); print }' FS=, file
99 01-jun-2018 sylvain team:::admin
52 01-dec-2018 sonia team
52 01-dec-2018 sonia team
25 01-jan-2019 sonia team
10 01-jan-2019 sylvain team:::admin
8 12-jun-2018 öle team:support
17 05-apr-2019 abhishek guest
```

The `gsub` function takes a regular expression to search, a replacement string and the variable containing the text to be modified in place. If that later is missing, $0 is assumed.

F. Working with external commands in AWK

Another great feature of AWK is you can easily invoke external commands to process your data. There are basically two ways of doing it: using the `system` instruction to invoke a program and letting it intermixing its output in the AWK output stream. Or using a pipe so AWK can capture the output of the external program for finer control of the result.

Those may be huge topics by themselves, but here are few simple examples to show you the power behind those features.

21. Adding the date on top of a file

```
awk 'BEGIN { printf("UPDATED: "); system("date") } /^UPDATED:/ { next }
UPDATED: Thu Feb 15 00:31:03 CET 2018
CREDITS,EXPDATE,USER,GROUPS
99,01 jun 2018,sylvain,team:::admin
52,01    dec   2018,sonia,team
52,01    dec   2018,sonia,team
25,01    jan   2019,sonia,team
10,01 jan 2019,sylvain,team:::admin
8,12    jun   2018,öle,team:support



17,05 apr 2019,abhishek,guest
```

In that AWK program, I start by displaying the work UPDATED. Then the program invokes the external date command, which will send its result on the output right after the text produced by AWK at that stage.
The rest of the AWK program just remove an update statement eventually present in the file and print all the other lines (with the rule `1`).

Notice the `next` statement. It is used to abort processing of the current record. It is a standard way of ignoring some records from the input file.

22. Modifying a field externally

For more complex cases, you may need to consider the `| getline VARIABLE` idiom of AWK:

```
awk '+$1 { CMD | getline $5; close(CMD); print }' CMD="uuid -v4" FS=, OF
99,01 jun 2018,sylvain,team:::admin,5e5a1bb5-8a47-48ee-b373-16dc8975f725
52,01    dec   2018,sonia,team,2b87e9b9-3e75-4888-bdb8-26a9b34facf3
52,01    dec   2018,sonia,team,a5fc22b5-5388-49be-ac7b-78063cbbe652
25,01    jan   2019,sonia,team,3abb0432-65ef-4916-9702-a6095f3fafe4
10,01 jan 2019,sylvain,team:::admin,592e9e80-b86a-4833-9e58-1fe2428aa2a2
8,12    jun   2018,öle,team:support,3290bdef-fd84-4026-a02c-46338afd4243
17,05 apr 2019,abhishek,guest,e213d756-ac7f-4228-818f-1125cba0810f
```

This will run the command stored in the `CMD` variable, read the first line of the output of that command, and store it into the variable `$5`.

Pay special attention to the close statement, crucial here as we want AWK to create a new instance of the external command each time it executes the `CMD | getline` statement. Without the close statement, AWK would instead try to read several lines of output from the same command instance.

23. Invoking dynamically generated commands

Commands in AWK are just plain strings without anything special. It is the *pipe operator* that triggers external programs execution. So, if you need, you can dynamically construct arbitrary complex commands by using the AWK string manipulation functions and operators.

```
awk '+$1 { cmd = sprintf(FMT, $2); cmd | getline $2; close(cmd); print }
99 2018-06-01 sylvain team:::admin
```

```
52 2018-12-01 sonia team
52 2018-12-01 sonia team
25 2019-01-01 sonia team
10 2019-01-01 sylvain team:::admin
8 2018-06-12 öle team:support
17 2019-04-05 abhishek guest
```

We have already met the `printf` function. `sprintf` is very similar but will return the built string rather than sending it to the output

24. Joining data

To show you the purpose of the close statement, I let you try out that last example:

```
awk '+$1 { CMD | getline $5; print }' CMD='od -vAn -w4 -t x /dev/urandom
99 01 jun 2018 sylvain team:::admin  1e2a4f52
52 01    dec   2018 sonia team  c23d4b65
52 01    dec   2018 sonia team  347489e5
25 01    jan   2019 sonia team  ba985e55
10 01 jan 2019 sylvain team:::admin  81e9a01c
8 12    jun   2018 öle team:support  4535ba30
17 05 apr 2019 abhishek guest  80a60ec8
```

As the opposite of the example using the `uuid` command above, there is here *only one* instance of `od` launched while the AWK program is running, and when processing each record, we read one more line of the output of that *same* process.

Conclusion

That quick tour of AWK certainly can't replace a full-fledged course or tutorial on that tool. However, for those of you that weren't familiar with it, I hope it gave you enough ideas so you can immediately add AWK to your toolbox.

On the other hand, if you were already an AWK aficionado, you might have found here some tricks you can use to be more efficient or simply to impress your friends.

However, I do not pretend been exhaustive. So, in all cases, don't hesitate to share your favorite AWK one-liner or any other AWK tips using the comment section below!