**Hello! AWK**

Awk is a powerful tools in the commandline used for processing the rows and columns of a flat texy file. Awk has built in string functions and associative arrays. Awk supports most of the operators, conditional blocks, and loops available in C language. You may want to know, what Awk stands for? It comes from the surnames of its authors *"Aho, Weinberger, and Kernighan"*. AWK was created at Bell Labs in the 1970s. It is pronounced the same as the name of a bird called auk. The GNU implementation of awk is called gawk.

This tutorial will give you **just enough** knowledge to read and understand this book, to be a master on AWK, you need to explore relevant literature referenced at end of this book.

The AWK language is a fully data-driven scripting language consisting of a set of actions to be taken against streams of textual data - either run directly on files or used as part of a pipelinefor purposes of extracting or transforming text, such as producing formatted reports.

The very basic syntax of AWK:

**awk 'BEGIN {start-action} {action} END {stop-action}' filename**

Note that the actions in the *begin block* are performed before processing the file and the actions in the *end block* are performed after processing the file. The rest of the actions are performed while processing the file!

It can be also written as:

**awk '/search pattern1/ {Actions}**

**/search pattern2/ {Actions}' file**

In the above AWK syntax:

- search pattern is a regular expression;
- Actions are the statement(s) to be performed;
- several patterns and actions are possible in AWK;
- a file is an input file; and
- single quotes around program is to avoid shell not to interpret any of its special characters.

# AWK Built-in Variables

The following tables lists some important AWK builtin variables:

| Variable | Description |
| --- | --- |
| `ARGC`, `ARGV` | command-line arguments |
| `FILENAME` | name of the file that awk is currently reading. |
| `FNR` | current record number in the current file, incremented on new records read |
| `NF` | number of fields in the current input record |
| `NR` | number of input records processed since the beginning of the program's execution |

| | |
|---|---|
| `RLENGTH` | length of the substring matched by the AWK's `match` function |
| `RS` | input record separator (default: newline) |
| `OFS` | output field separator (default: blank) |
| `ORS` | output record separator (default: newline) |
| `OFMT` | output format for numbers (default: %.6g) |
| `ENVIRON` | array of environment variables; subscripts are names. |

## AWK built-in functions

AWK has the mathematical functions like `exp`, `log`, `sqrt`, `sin`, `cos`, `atan2`, etc. built-in, other built-in functions are:

- `length` the length of its argument taken as a string, or of`$0` if no argument.
- `rand` random number between `0` and `1`
- `srand` sets seed for rand and returns the previous seed.
- `int` truncates to an integer value
- `substr(s, m, n)` the `n`-character substring of `s` that begins at position `m` counted from `1`.
- `index(s, t)` the position in `s` where the string `t` occurs, or `0` if it does not.
- `match(s, r)` the position in `s` where the regular expression `r` occurs, or `0` if it does not. The variables `RSTART` and `RLENGTH` are set to the position and length of the matched string.
- `split(s, a, fs)` splits the string `s` into array elements `a[1], a[2], ..., a[n]`, and returns `n`. The separation is done with the regular expression `fs` or with the field separator `FS` if `fs` is not given. An empty string as field separator splits the string into one array element per character.
- `sub(r, t, s)` substitutes `t` for the first occurrence of the regular expression `r` in the string `s`. If `s` is not given, `$0` is used.
- `gsub` same as sub except that all occurrences of the regular expression are replaced; sub and gsub return the number of replacements.
- `sprintf(fmt, expr, ... )` the string resulting from formatting `expr ...` according to the `printf` format `fmt`.
- `system(cmd)` executes cmd and returns its exit status.
- `tolower(str)` returns a copy of str with all upper-case characters translated to their corresponding lower-case equivalents.
- `toupper(str)` returns a copy of str with all lower-case characters translated to their corresponding upper-case equivalents.

# AWK useful examples

The best way to learn AWK is probably looking at some examples in a given context, let us create file `data.txt` file which has the following content:

*File content: data.txt*

| Months | Water | Gas | Elec | Phone |
|---|---|---|---|---|
| Jan | 647 | 2851 | 3310 | 1200 |
| Feb | 287 | 3544 | 25500 | 1000 |
| Mar | 238 | 3212 | 21900 | 1710 |
| Apr | 468 | 6986 | 35000 | 2000 |
| May | 124 | 1097 | 96300 | 13000 |

*Example 1. AWK `print` function*

By default Awk prints every line from the file.

**awk -F, '{print;}' data.txt | csvlook**

Action `print` with out any argument prints the whole line by default. So it prints all the lines of the file with out fail. Note that the actions need to be enclosed with in the braces.

### *Example 2. AWK `print` specific field*

```
awk -F, '{print $1,$2}' data.txt
```

### *Example 3. AWK's `BEGIN` and `END` Actions*

```
awk -F, 'BEGIN {print "Period\tBill1\tBill2";}

{print $1,"\t",$2,"\t",$3,"\t",$NF;}

END{print "END\n----------"; }' data.txt
```

Here, the `actions` specified in the `BEGIN` section executed before AWK starts reading the lines from the input and `END` actions will be performed after completing the reading and processing the lines from the input.

### *Example 4. AWK fields variable (`$1`, `$2` and so on)*

```
#!/bin/bash

echo -n "Field sum: Water + Gas + Electtricty + Phones = "

awk -F "," '{

        if(FNR == 1){

                next;

        }

        Water=$2;

        Gas=$3;

        Electricity=$4;

        Phones=$5;


        fields_sum=Water + Gas + Electtricty + Phones;


        total +=fields_sum;
```

```
} END { print total; }' data.txt
```

Let's consider we want to find a total of al bills in all months in the data. We then create the following script:

Note that it's a bash script that calls `awk` from inside and we have used `FNR` to detect the first row which we want to avoid in the sum calculation.

*Example 5. AWK built-in variables*

As mentioned earlier, the built-in variable `$NF` represents number of field, in this case last field (5):

```
awk -F, '{print $1,$NF;}' data.txt
```

*Example 6. AWK fields comparison `>`*

Let's now find the months with water `bills > 500`:

```
awk -F, '$2 > 500' data.txt
```

## Self-contained AWK scripts example

In Linux systems self-contained AWK scripts can be constructed using. For example, a script that prints the content of a given file may be built by creating a file named `printfile.awk` with the following content:

```
awk -f printfile.awk  data.txt
```

The `-f` flag tells AWK that the argument that follows is the file to read the AWK program from.

# Hello! SED - Stream Editor

**Stream EDitor (SED)** is an important text-processing utilities on GNU/Linux. It uses a simple programming language and is capable of solving complex text processing tasks with few lines of code. This easy, yet powerful utility makes GNU/Linux more interesting.

SED can be used in many different ways, such as:

- Text substitution,
- Selective printing of text files,
- In-a-place editing of text files,
- Non-interactive editing of text files, and many more.

This tutorial will give you **just enough** knowledge to read and understand this book, to be a master on the SED, GREP and Find command, you need to explore relevant literature referenced at end of this book.

**Sed works as follows:** it reads from the standard input, one line at a time. for each line, it executes a series of editing commands, then the line is written to `STDOUT`.

An example which shows how it works : we use the `s` sommand. `s` means "substitute" or search and replace. The format is

```
sed s/regular-expression/replacement text/{flags}
```

In the example below, we have used `g` as a flag, which means "replace all matches" (global replacement):

```
$ cat datafile.txt

 I have a big data!


$ sed -e 's/big/small/g' -e 's/data/list/g' datafile.txt

 I have a small list!
```

Let's try to learn what happened.

**Step 1**, sed read in the line of the file and executed

```
 s/big/data/g
```

which produced the following text:

```
I have a small data!
```

**Step 2**, then the second replacement command (`'s/data/list/g'`) was performed on the edited line and the result was:

I have a small list!

# SED substitutions

The format for the substitute command is as follows:

```
[address1[ ,address2]]s/pattern/replacement/[flags]
```

The flags can be any of the following:

- `n` replace `n`th instance of pattern with replacement
- `g` replace all instances of pattern with replacement
- `p` write pattern space to STDOUT if a successful substitution takes place
- `w file` Write the pattern space to file if a successful substitution takes place
- `i` match REGEXP in a case-insensitive manner.

We can use different delimiters ( one of `@ % ; :` ) instead of `/`. If no flags are specified the first match on the line is replaced. note that we will almost always use the `s` command with either the `g` flag or no flag at all.

If one address is given, then the substitution is applied to lines containing that address. An address can be either a regular expression enclosed by forward slashes `/regex/` , or a line number . The `$` symbol can be used in place of a line number to denote the last line. If two addresses are given separated by a comma, then the substitution is applied to all lines between the two lines that match the pattern.

Example 1: substitute only third occurrence of a word `sed s//3`

```
$ sed 's/Data/Big-Data/3' datafile
```

Example 2: print and write to a file `sed s//gpw`

```
$ sed -n 's/Data/Big-Data/gpw output' datafile
```

*Some important SED options*

**Option:** `-n`, `--quiet` or `--silent`

The `"-n"` option will not print anything unless an explicit request to print is found:

```
sed -n 's/PATTERN/&/p' file
```

**Option:** `-e`

Combines multiple commands:

```
sed -e 's/a/A/' -e 's/b/B/' file
```

**Option:** `-f`

If you have a large number of sed commands, you can put them into a file and use

```
sed -f sedscript file
```

**Option:** `-r`

Extended regular expressions (ERE) have more power, but SED them normal characters. Therefore you must explicitly enable this extension with a command line option.

```
% echo "123 abc" | sed -r 's/[0-9]+/& &/'
123 123 abc
```

**Option:** `-i`

Substitutions a re performed in-place, on the file which was fed to SED:

```
sed -i 's/^/\t/' *.txt
```

# SED and regular expressions

Let's learn the use of `SED + REGEX` by using a couple of practical exmaples:

**Example 1:** match patterns and REGEX (`\!.*`)

```
$ sed '/\!/s/\!.*//g' datafile
```

In this example, if the line matches with the pattern “!“, then it replaces all the characters from “!” with an empty char.

**Example 2:** use REGEX to delete the last x chacaters

```
$ sed 's/...$//' datafile
```

This sed example deletes last 3 characters from each line.

**Example 3:** use REGEX to eliminate comments (#)

```
$ sed -e 's/#.*//'  datafile
```

This sed example deletes last 3 characters from each line.

**Example 4:** use REGEX and eliminate Comments (#) as well as the empty lines

```
$ sed -e 's/#.*//;/^$/d'   datafile
```

This sed example deletes last 3 characters from each line.

# SED delete, print and grouping

## SED delete

A useful command that deletes line that matches the restriction: "d." For example, if you want to chop off the header of a mail message, which is everything up to the first blank line, use:

```
$ sed '1,/^$/d' file
```

## SED print

If sed was not started with an `"-n"` option, the `"p"` command will duplicate the input.

```
sed '/^$/ p' file
```

Adding the "`-n`" option turns off printing unless you request it.

## SED grouping

The curly braces, `"{"` and `"}"` are used to group the commands.

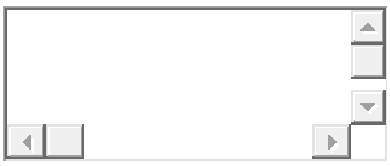
Previously, we have showed you how to remove comments starting with a `"#."` If you wanted to restrict the removal to lines between special `"begin"` and `"end"` key words, we use:

```
sed -n '
  /begin/,/end/ {
      s/#.*//
      s/[ ^I]*$//
      /^$/ d
      p
  }
' file
```

## What is GREP?

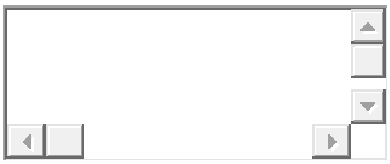
The command `grep` is a small utility for searching plain-text data sets for lines matching a regular expression. Its name comes from the globally search a regular expression and print.

A simple example of a common usage of grep is the following, which searches the file `colors.txt` for lines containing the text string `blue`:



```
$ grep "red" colors.txt
```

The `v` option reverses the sense of the match and prints all lines that do not contain `blue`, as in this example.



```
$ grep -v "blue" colors.txt
```

The `i` option in grep helps to match words that are case insensitive, as shown in below example.

```
$ grep -i "bLuE" colors.txt
```

The `n` option identifies the lines where matches occured:

```
$ grep -n "orange" colors.txt
4: Orange color
6: Ornage company
```

## GREP and regular expressions

While grep supports a handful of regular expression commands, it does not support certain useful sequences such as the + and ? operators. If you would like to use these, you will have to use extended grep (egrep).

The Following command illustrates the ?, which matches 1 or 0 occurences of the previous character `w`:

```
$ grep  "yellow?" colors.txt
```

`egrep` example:

```
$ egrep  "red|yellow" colors.txt
```

Note that `grep` does not do the pipe (|), which funcitons as an `"OR"` in the expression.