# Command Line Case Studies: awk

ZENO KOLLER - (EX)CHEFREDAKTOR AND COMMANDLINE NINJA

**In "Command Line Case Studies", we want to showcase some command line tools that are available on Unix machines. The goal is to not only repeat examples that one might find in a tutorial, but also motivate the use of the tool with a real-life application. More often than not, when presented with some computer trickery, you might think, "Neat!", but end up not incorporating the skill in your own workflow. Here, real-life applications could help you see the light.**

If you still use computers with a desktop operating system, you've probably encountered ***$STUPID_COMPUTER_PROBLEM*** before. Here, I use this term for computer-related things that, at first sight, seem more complicated than they need to be. Say you want to convert a file to some other format and the (GUI) software at hand does not support it. In frustration, one is often tempted to think:

- I can formulate the problem and, with high confidence, the solution. Why can this universal computing machine not simply do it for me?

Of course, this way of thinking leads to a dead end. So, continuing the line of thought:

- Surely, there must be some command-line wizardry that does the job. If only I knew the required tools by heart…

At this point, you have two options: First, you could try to find some domain-specific GUI tool that more often than not is just an interface for an underlying, more general command line tool. It either costs a lot or bombards you with ads. Or, you can take some time and figure out how to do it more intelligently and learn something.

Let's face it: While studying computer science at ETH teaches you a lot, usage of command line utilities does not belong to the syllabus. To be fair, I think this is not something that can effectively be taught in a university course[1]. There are biblical volumes such as Unix Power Tools[2] that teach you all the ins and outs of the shell. While the examples given in those books are often neat, they are just examples. The few commands I regularly use on the shell, either somebody showed me in person or I ran into ***$STUPID_COMPUTER_PROBLEM*** and was motivated enough to do some research on how to solve this mess using the shell. For me, this motivation is essential – and this is what I want to achieve with this article, or hopefully, head(series).

## Introduction: awk

Let's start with AWK. More of a programming language than simply a utility, it is designed as a quick-and-dirty way to process text files and extract data and reports from them. The ***awk*** command interprets whatever AWK program you supply. If your program surpasses some amount of complexity, you're better off using spreadsheet software or a "real" programming language, but AWK is still fun like that.

Let's start with a simple example. Suppose you live in a shared flat and track each flatmate's expenses for food as well as the number of times they've eaten a meal in the month of December in a text file ***december.tsv***.

```
Zeno    118.30    18
Maria    126.10    24
Chantal    149.25    29
Robin     46.30     9
```

With a simple AWK program, we can find out how many times Robin has eaten this month.

```
> awk '$1 == "Robin" { print $3 }'
december.tsv
9
```

The single-quoted string is the actual program. To understand why it works this way, we have to understand how AWK operates. A usual AWK program has one or more pattern-action statements:

```
pattern { action }
```

AWK reads the data file line by line and splits each line into fields that can be accessed with ***$1, $2, ….*** By default, fields are separated by spaces or tabs. In this example, ***$1*** denotes the first field. If the line matches a ***pattern***, the corresponding action is executed on this line. If the ***action*** is omitted, AWK prints the whole line. The pattern may also be omitted, in which case the action is executed for every line, as in the next example, where we show how much our four flatmates spent in total.

```
> awk '{ sum = sum + $2 } END { print
sum }' december.tsv
439.95
```

Here we see that AWK also handles variables. You can assign numbers or strings to them, and they're initialized to the empty string or zero, as soon as numerical usage is recognized. Notice the ***END*** keyword: It is a special kind of pattern. The associated action is executed after all the lines have been read. Similarly, there is ***BEGIN***, where AWK executes the associated action once before starting to read lines.

Keep in mind that I am only scratching the surface here. There is much more to this language, such as loops, control flow statements and the like. If you'd like to know the exact specification, I refer you to the man page, the Wikipedia page[3] or the book The AWK Programming Language[4]. To showcase some of the available features, I'm finishing this introduction with an AWK program that computes each person's share of the expenses, in proportion to the meals they've eaten. Here, we can also see the limits of the language. We'd like to first sum the two numeric columns, but still use the values from each line. We could read the file once, sum up the columns and store the values of each line in an array and output the report using a for loop in an ***END*** statement.

➔

```
{
    expense_sum = expense_sum + $2
    meals_sum = meals_sum + $3

    # `NR` is the variable for the
current line number
    name[NR] = $1
    expense[NR] = $2
    meals[NR] = $3
}
END {
    for (i = 1; i <= NR; i++) {
        share = (expense_sum *
(meals[i] / meals_sum))

        imbalance = expense[i] -
share

        if (imbalance > 0) {
            printf("%s receives CHF
%2.1f0\n", name[i], imbalance)
        } else {
            printf("%s owes CHF
%2.1f0\n", name[i], -1 * imbalance)
        }
    }
}
```

An alternative way is to go through the file twice with a little trick. Variable ***FNR*** stores the line number in the current file, while ***NR*** is not reset when opening another file. Neither are the accumulated counts from the first file. The pattern ***NR == FNR*** will only evaluate to true for the first file and the ***next*** keyword will go to the next line, which skips the second action for the first file.

```
NR == FNR {
    # only executed for the first file

    expense_sum = expense_sum + $2
    meals_sum = meals_sum + $3
    next
}
{
    # only executed for the second
file

    share = (expense_sum * ($3 /
meals_sum))
    imbalance = $2 - share

    if (imbalance > 0) {
        printf("%s receives CHF
%2.1f0\n", $1, imbalance)
    } else {
        printf("%s owes CHF
%2.1f0\n", $1, -1 * imbalance)
    }
}
```

For this code to work, we have to invoke the program while telling AWK to read the data file twice:

```
> awk -f iou.awk december.tsv
december.tsv
Zeno receives CHF 19,20
Maria owes CHF 5,70
Chantal owes CHF 10,10
Robin owes CHF 3,40
```

Here you also see how to execute an AWK program from a file. You can decide for yourself whether you like version one or two better, or none of those.

➔

„Unsere Softwarelösungen setzen neue Standards in der Sensorik.“
Eduard Rudi,
Software Engineer

**„Become part of the Sensirion success story”.** Wollen Sie Ihrer Karriere den entscheidenden Kick geben und sich neuen Herausforderung stellen? Dann heissen wir Sie herzlich willkommen bei Sensirion.

Sensirion steht für Hightech, Innovation und Spitzenleistungen. Wir sind der international führende Hersteller von hochwertigen Sensor- und Softwarelösungen zur Messung und Steuerung von Feuchte, Gas- und Flüssigkeitsdurchflüssen. Unsere Sensoren werden weltweit millionenfach in der Automobilindustrie, der Medizintechnik und der Konsumgüterindustrie eingesetzt und tragen zur stetigen Verbesserung von Gesundheit, Komfort und Energieeffizienz bei. Mit unserer Sensorik liefern wir damit einen aktiven Beitrag an eine smarte und moderne Welt.

Schreiben Sie Ihre eigenen Kapitel der Sensirion Erfolgsgeschichte und übernehmen Sie Verantwortung in internationalen Projekten. Stimmen Sie sich auf www.sensirion.com/jobs auf eine vielversprechende Zukunft ein.

www.sensirion.com/jobs

SENSIRION
THE SENSOR COMPANY

## Use Case: Counting Steps

Okay, let's stop pretending we track our expenses with text files and get to another use case – one that actually occurred in real life.

The iPhone (5s or newer) features a motion processor that also serves as a pedometer. It's a neat feature; you can see how far you've walked. The pre-installed Health app provides only little reporting, such as the daily average over weeks, months or years. What if I want to ask the data my own questions? By sifting through a million applications in the App Store to find one that does what you want? Good luck with that. Fortunately, you can export the data as an ***`export.xml`*** file[5]. The file contains all kinds of health-related data the phone might have aggregated, for instance, the count of flights of stairs climbed. Here, we are only interested in the steps. Looking at the entries, we can see that the steps seem to be entries of type ***`HKQuantityTypeIdentifierStepCount`***.

```
<Record type="HKQuantityTypeIdentifier
StepCount" sourceName="Zeno's iPhone
SE" sourceVersion="9.1" unit="count"
creationDate="2015-11-01 17:13:51
+0100" startDate="2015-11-01 16:21:52
+0100" endDate="2015-11-01 16:26:53
+0100" value="766"/>
<Record type="HKQuantityTypeIdentifier
StepCount" sourceName="Zeno's iPhone
SE" sourceVersion="9.1" unit="count"
creationDate="2015-11-01 17:13:51
+0100" startDate="2015-11-01 16:26:53
+0100" endDate="2015-11-01 16:32:07
+0100" value="755"/>
```

Part of the entries are omitted for brevity[6]. We can see that the steps are counted for periods of a few minutes, which is not very practical for three years worth of data. We'd rather aggregate the data by day instead. Now we could, for example, write some Python code, import some XML library, open the file and sum the steps for each day. But with that approach, you have the overhead of finding the XML library, reading the docs, importing the file. If you wanted to do further processing, this would be fine, but for now, we just want the daily counts. There are also command line tools for parsing XML, which are a better choice for more complicated data formats, but here we can just use AWK because the entries are in a fairly simple, consistent format. Instead of spaces or tabs, we can define custom field separators using the ***`-F`*** command. With a regular expression, we can tell AWK to use both quotes and spaces as separators. First, we'd like to find the columns that contain the date and the steps.

```
awk -F'\"| ' '/ <Record type="HKQuant
ityTypeIdentifierStepCount"/  { for (i
= 1; i <= NF; i++){ printf("Field %d:
%s\n",i,$i)}; exit }' export.xml
```

Here, the pattern ***`/ <Record type="HKQuantityTypeIdentifierStepCount"/`*** has the form of a regular expression (we only want to match lines that contain the step count). The corresponding action prints each field number and what it contains. ***`exit`*** stops the program after one action. By inspecting the output

```
[...]
Field 26: creationDate=
Field 27: 2015-09-18
[...]
Field 41: value=
Field 42: 17
[...]
```

We can see that the values we're interested in are in columns 27 and 42, respectively. With this information, we can now write the program that sums up the steps for each day.

```
BEGIN { FS = "\"| " }
/<Record type="HKQuantityTypeIdentifie
rStepCount"/ {
    if (date != $27) {
        print date, steps

        date = $27
        steps = 0
    }

    steps = steps + $42
}

END { print date"\t"steps }
```

Executing it with

```
awk -f sum_daily_steps.awk export.xml
> daily_steps.tsv
```

stores the steps in a file. The output looks as follows:

```
2014-09-26    6543
2014-09-27    8397
2014-09-28    7687
2014-10-01    6284
2014-10-02    10301
...
```

Now that we extracted the daily steps from the file, we're ready for further processing to extract some interesting information. For example, to view the five days with the most steps. ***r*** sorts in reverse order, ***n*** sorts numerically, ***k2*** uses the second column as a key.

```
> sort -rnk2 daily_steps.tsv | head
-5
2015-03-18 51905
2015-02-22 41505
2016-04-30 33403
2016-07-21 32690
2015-07-04 30005
```

The top two entries have been recorded using an iPhone 5s, where either the sensor or the recognition software was not as accurate yet, which resulted in Bus and Car rides being counted as steps. The third entry is from a day spent mountain biking, another false entry! The fourth and fifth entries are finally “real” step counts, I was hiking both of those days.

We could also find out what is the longest streak of days with over 10'000 steps.

```
{
    if ($2 < 10000) {
        days = 0
        steps = 0
    } else {
        if (days == 0) {
            date = $1
        }

        days = days + 1
        steps = steps + $2
    }

    if (max_days < days) {
        max_days = days
        max_steps = steps
        max_date = date
    }
}
END { print "Longest streak started
on", max_date, ", lasted", max_days,
"days and resulted in", max_steps,
"steps" }
```

➔

This one is fairly simple. The output will be

```
> awk -f streak.awk daily_steps.tsv
Longest streak started on 2015-07-22,
lasted 7 days and resulted in 115949
steps
```

There are lots and lots more simple, fun questions. You could aggregate the data by months or specific weekdays. What is the weekday with the most activity? Do I really move more in the summer than in the winter? What has been the laziest week? Et cetera. Maybe you want to try those for yourself? For further processing, you could also import the file into Excel, Matlab or whatever you like.

If you've read this far and found this article way too basic, you might want to contribute your own command line case study as an article! Feel free to write to visionen@vis.ethz.ch.

## Footnotes

[1] Or is it? I would enrol in Command Line Case Studies Seminar in the blink of an eye.

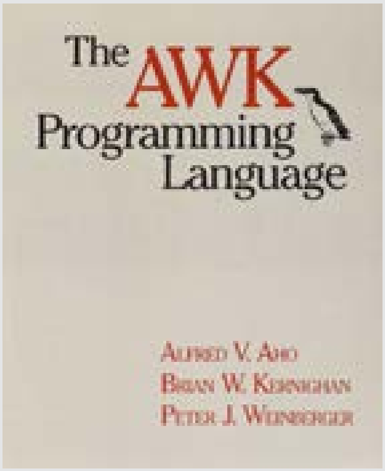
[2] Powers, Shelley. UNIX power tools. O'Reilly Media, Inc., 2003

[3] https://en.wikipedia.org/wiki/AWK

[4] Aho, Alfred V., Brian W. Kernighan, and Peter J. Weinberger. The AWK programming language. Addison-Wesley Longman Publishing Co., Inc., 1987

[5] How to export your health data on iOS 10 - https://www.igeeksblog.com/how-to-export-import-health-data-in-ios-10/

[6] Also omitted for brevity, but you will notice if you try to do the same thing with your health data: Between 2014 and 2016, Apple used three different record formats. I handled this in my script by using more specific patterns, one for each format.



The AWK language was created in the 1970s at Bell Labs by Alfred Aho, Peter Weinberger and Brian Kernighan (the co-author of C). The name AWK is just the first letters of the creator's surnames. The trio also wrote the book The AWK Programming Language4, which sports the auk (which is how you pronounce AWK), a type of bird, on its cover. I first thought the bird represents socially AWKward penguin. Thanks to Wikipedia, I now know better.