Awk One-Liners Explained

by

@pkrumins
Peteris Krumins
peter@catonmat.net
http://www.catonmat.net
good coders code, great reuse

Contents

Preface

Thanks!

Thank you for purchasing my "Awk One-Liners Explained" e-book! This is my first e-book that I have ever written and I based it on article series "Famous Awk One-Liners Explained" that I wrote on my www.catonmat.net blog. I went through all the one-liners in the articles, improved them, fixed a lot of mistakes, added an introduction to Awk one-liners and two new chapters. The two new chapters are Awk Special Variables that summarizes some of the most commonly used Awk variables and Idiomatic Awk that explains what idiomatic Awk is.

You might wonder why I called the article series "famous"? Well, because I based the articles on the famous awk1line.txt file by Eric Pement. This file has been circulating around Unix newsgroups and forums for years and it's very popular among Unix programmers. That's how I actually learned the Awk language myself. I went through all the one-liners in this file, tried them out and understood how they exactly work. Then I thought it would be a good idea to explain them on my blog, which I did, and after that I thought, why not turn it into a book? That's how I ended up writing this book.

I have also planned writing two more books called "Sed One-Liners Explained" and "Perl One-Liners Explained". The sed book will be based on Eric Pement's sed1line.txt file and "Famous Sed One-Liners Explained" article series and the Perl book will be based on my "Famous Perl One-Liners Explained" article series. I am also going to create perl1line.txt file of my own. If you're interested, subscribe to my blog and follow me on Twitter. That way you'll know when I publish all of this!

Credits

I'd like to thank Eric Pement who made the famous awk1line.txt file that I learned Awk from and that I based this book on. I'd also like to thank waldner and pgas from #awk channel on FreeNode IRC network for always helping me with Awk, Madars Virza for proof reading the book before I published it and correcting several glitches, Antons Suspans for proof reading the book after I published it, Abraham Alhashmy for giving advice on how to improve the design of the book, everyone who commented on my blog while I was writing the Awk one-liners article series, and everyone else who helped me with Awk and this book.

One

Introduction

1.1 Awk One-Liners

Knowing Awk makes you really powerful when working in the shell. Check this out, suppose you want to print the usernames of all users on your system. You can do it very quickly with this one-liner:

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

This is really short and powerful, isn't it? As you know, the format of `/etc/passwd` is colon separated:

```
root:x:0:0:1667:/root:/bin/bash
```

The one-liner above says: Take each line from `/etc/passwd`, split it on the colon `-F:` and print the first field `$1` of each line.

Here are the first few lines of output when I run this program on my system:

```
root
bin
daemon
adm
lp
sync
...
```

Exactly what I expected.

Now compare it to a C program that I just wrote that does the same:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_LINE_LEN 1024

int main() {
    char line[MAX_LINE_LEN];
    FILE *in = fopen("/etc/passwd", "r");
    if (!in) exit(EXIT_FAILURE);

    while (fgets(line, MAX_LINE_LEN, in) != NULL) {
        char *sep = strchr(line, ':');
        if (!sep) exit(EXIT_FAILURE);
        *sep = '\0';
        printf("%s\n", line);
    }
    fclose(in);
    return EXIT_SUCCESS;
}
```

This is much longer and you have to compile the program, only then you can run it. If you make any mistakes, you have to recompile again.

That's why one-liners are called one-liners. They are short, easy to write and they do one and only one thing really well. I am pretty sure you're starting to see how mastering Awk and one-liners can make you much more efficient when working in the shell, with text files and with computers in general.

Here is another one-liner, this one numbers the lines in some file:

```
awk '{ print NR ". " $0 }' somefile
```

Isn't this beautiful? The `NR` special variable keeps track of current line number so I just print it out, followed by a dot and `$0` that, as you'll learn, contains the whole line. And you're done.

I know that a lot of my book readers would argue that Perl does exactly the same, so why should you learn Awk? My answer is very simple, yes, Perl does exactly the same, but why not be the master of the shell? Why not learn Awk, sed, Perl and other utilities? Besides Perl was created based on ideas from Awk, so why not learn Awk to see how Perl evolved. That gives you a unique perspective on programming languages, doesn't it?

Overall, this book contains 70 well explained one-liners. Once you go through them, you should have a really good understanding of Awk and you'll be the master shell problem solver. Enjoy this book!

Two

Line Spacing

2.1 Double-space a file

```
awk '1; { print "" }'
```

So how does this one-liner work? A one-liner is an Awk program and every Awk program consists of a sequence of pattern-action statements `pattern { action statement }`. In this case there are two statements `1` and `{ print "" }`. In a pattern-action statement either the pattern or the action may be missing. If the pattern is missing, the action is applied to every single line of input. A missing action is equivalent to `{ print }`. The first pattern-action statement is missing the action, therefore we can rewrite it as:

```
awk '1 { print }; { print "" }'
```

An action is applied to the line only if the pattern matches, i.e., pattern is true. Since `1` is always true, this one-liner translates further into two print statements:

```
awk '{ print }; { print "" }'
```

Every print statement in Awk is silently followed by the `ORS` – Output Record Separator variable, which is a newline by default. The first print statement with no arguments is equivalent to `print $0`, where `$0` is the variable holding the entire line (not including the newline at the end). The second print statement seemingly prints nothing, but knowing that each print statement is followed by `ORS`, it actually prints a newline. So there we have it, each line gets double-spaced.

We can also drop the semicolon and write it as:

```
awk '{ print } { print "" }'
```

We can do it because Awk parser is smart enough to understand that those are two separate actions.

And, of course, you can join two actions with the same pattern (in this case no pattern) into one:

```
awk '{ print; print "" }'
```

And just to play around a bit, you can also achieve the double-spacing of a file this way:

```
awk '{ print $0 "\n" }'
```

This one-liner appends the newline symbol `\n` to the whole line `$0` and prints it.

2.2 Another way to double-space a file

```
awk 'BEGIN { ORS="\n\n" }; 1'
```

`BEGIN` is a special kind of pattern, which is not tested against the input. It is executed before any input is read. This one-liner double-spaces the file by setting the `ORS` variable to two newlines. As I mentioned in the first one-liner, statement `1` gets translated to `{ print }`, and every print statement gets terminated with the value of `ORS` variable. As a result every line gets printed out with two newlines at the end, which double-spaces the file.

2.3 Double-space a file so that no more than one blank line appears between lines of text

```
awk 'NF { print $0 "\n" }'
```

The one-liner uses another special variable called `NF` – Number of Fields. It contains the number of fields (columns) on the current line. For example, a line "this is a test" has four fields and `NF` gets set to 4. The empty line does not have any fields so `NF` gets set to 0. Using `NF` as a pattern can effectively filter out empty lines. This one liner says: "If the line is not empty, print the whole line followed by newline." Now if there are any newlines after the current line, they get ignored and we get no more than one blank like after each line of text.

2.4 Triple-space a file

```
awk '1; { print "\n" }'
```

This one-liner is very similar to the first two. `1` gets translated into `{ print }` and the resulting Awk program is:

```
awk '{ print; print "\n" }'
```

It prints the line, then prints a newline followed by terminating `ORS`, which is newline by default.

Another way to do the same is to call `{ print "" }` twice:

```
awk '{ print } { print "" } { print "" }'
```

And since all three actions are applied to the same line, we can join them:

```
awk '{ print; print ""; print "" }'
```

Or we can just print the line `$0` and two newlines after it:

```
awk '{ print $0 "\n\n" }'
```

2.5 Join all lines

```
awk '{ printf "%s ", $0 }'
```

This one-liner uses the `printf` function instead of `print` to print the line. The `printf` function does formatted printing. You specify a template and arguments to the template and it prints it out. In this one-liner the template is `"%s "`, which means print out the first argument that is a string followed by a space. The argument is the whole line `$0`. So the one-liner just joins all lines by a space.

Another way to write the same is to modify `ORS` in the `BEGIN` block:

```
awk 'BEGIN { ORS=" " } { print }'
```

I explained the `ORS` variable in the first one-liner. It's a newline by default but here we modified it to be a space.

We can simplify this one-liner even further and change `{ print }` to something that is true, like `1`:

```
awk 'BEGIN { ORS=" " } 1'
```

Here is another trick. Instead of defining and modifying variables in the `BEGIN` block, you can use the `-v` command line argument:

```
awk -v ORS=" " 1
```

Holy cow this is awesome. The whole Awk program is just the statement `1`. And the `-v ORS=" "` did set the `ORS` variable to `" "` just like in the `BEGIN` block!

Three

Numbering and Calculations

3.1 Number lines in each file separately

```
awk '{ print FNR "\t" $0 }'
```

This Awk program appends the `FNR` – File Line Number predefined variable and a tab `\t` before each line. Construct `FNR "\t" $0` means concatenate value of FNR value with a tab with `$0` (whole line). The `FNR` variable contains the current line for each file separately. For example, if this one-liner was called on two files, one containing 10 lines, and the other 12, it would number lines in the first file from 1 to 10, and then resume numbering from 1 for the second file and number lines in this file from 1 to 12. `FNR` gets reset from file to file.

3.2 Number lines for all files together

```
awk '{ print NR "\t" $0 }'
```

This one works the same as the previous except that it uses `NR` – Line Number variable, which does not get reset from file to file. It counts the input lines seen so far. For example, if it was called on the same two files with 10 and 12 lines, it would number the lines from 1 to 22 (10 + 12).

3.3 Number lines in a fancy manner

```
awk '{ printf("%5d : %s\n", NR, $0) }'
```

This one-liner uses the `printf` function to number lines in a custom format. It takes format parameter just like a regular `printf` function in C. Note that `ORS` does not get appended at the end of `printf`, so we have to print the newline `\n` character explicitly. This one-liner right-aligns the line numbers, followed by a space and a colon, another space and the line.

If you wish to left-align the numbers, use the left-align flag for `printf` format specifier:

```
awk '{ printf("%-5d : %s\n", NR, $0) }'
```

3.4 Number only non-blank lines in files

```
awk 'NF { $0=++a " :" $0 }; { print }'
```

Awk variables are dynamic; they come into existence when they are first used. This one-liner pre-increments the variable `"a"` each time the line is non-empty, then it appends the value of this variable to the beginning of the line and prints it out.

It uses the `NF` variable as the pattern. As I explained in one-liner 2.3, `NF` is the number of fields (columns) on the line. When used as a pattern, the action is executed only if there is something in the line (the line is non-blank).

3.5 Count lines in files

```
awk 'END { print NR }'
```

`END` is another special kind of pattern, which is not tested against the input. It is executed when all the input has been exhausted. This one-liner outputs the value of `NR` special variable after all the input has been

consumed. `NR` contains total number of lines seen, so at the end of the file it's the total number of lines in the file.

3.6 Print the sum of fields in every line

```
awk '{ s = 0; for (i = 1; i <= NF; i++) s = s+$i; print s }'
```

Awk has some features of the C language, like the `for (;;) { ... }` loop. This one-liner loops over all fields in a line (there are `NF` fields in a line), and adds the result in variable `s`. Then it prints the result out and proceeds to the next line.

The most interesting part in this one-liner is the `s = s+$i`. The `$i` means "contents of field i". Awk allows indexing fields indirectly, so when `i` is 1, `$i` is the value of the 1st field, when `i` is 2, `$i` is the value of 2nd field, etc. The one-liner loops from field 1 to field `NF` and `s = s+$1` sums the values of each field. Before each line the variable `s` is reset to 0 by `s = 0`.

3.7 Print the sum of fields in all lines

```
awk '{ for (i = 1; i <= NF; i++) s = s+$i };
END { print s+0 }'
```

This one-liner is basically the same as previous one-liner, except that it prints the sum of all fields. Notice how it did not initialize variable `s` to 0. It was not necessary as variables come into existence dynamically. Also notice how it calls `print s+0` and not just `print s`. This is necessary in case there are no fields. If there are no fields, `s` never comes into existence and is undefined. Printing an undefined value does not print anything (i.e. it prints just the `ORS`). Adding a `0` does a mathematical operation and `undef+0` = `0`, so in case all lines are empty, it prints the correct value `0`.

3.8 Replace every field by its absolute value

```
awk '{ for (i = 1; i <= NF; i++)
if ($i < 0) $i = -$i; print }'
```

This one-liner uses two other features of C language, namely the `if (...) { ... }` conditional statement and omission of curly braces. It loops over all fields in a line and checks if any of the fields is less than 0. If any of the fields is less than 0, then it just negates the field to make it positive. Fields can be addresses indirectly by a variable. For example, `i = 5; $i = 'hello'`, sets the fifth field to the string `hello`.

Here is the same one-liner rewritten with curly braces for clarity. The `print` statement gets executed after all the fields in the line have been replaced by their absolute values.

```
awk '{
  for (i = 1; i <= NF; i++) {
    if ($i < 0) {
      $i = -$i;
    }
  }
  print
}'
```

3.9 Count the total number of fields (words) in a file

```
awk '{ total = total + NF }; END { print total+0 }'
```

This one-liner matches all the lines and keeps adding the number of fields in each line. The number of fields seen so far is kept in a variable named `total`. Once the input has been processed, the special pattern `END { ... }` is executed, which prints the total number of fields. See one-liner 3.7 for explanation of why we `print total+0` in the `END` block.

3.10 Print the total number of lines containing word "Beth"

```
awk '/Beth/ { n++ }; END { print n+0 }'
```

This one-liner has two pattern-action statements. The first one is `/Beth/ { n++ }`. A pattern between two slashes is a regular expression. It matches all lines containing pattern `Beth` (not necessarily the word "Beth", it could as well be "Bethe" or "theBeth1689"). When a line matches, variable `n` gets incremented by one. The second pattern-action statement is `END { print n+0 }`. It is executed when the file has been processed. Note the `+0` in `print n+0` statement. It forces `0` to be printed in case there were no matches (`n` was undefined). Had we not put `+0` there, an empty line would have been printed as the result.

3.11 Find the line containing the largest (numeric) first field

```
awk '$1 > max { max=$1; maxline=$0 };
END { print max, maxline }'
```

This one-liner keeps track of the largest number in the first field `$1` in variable `max` and the corresponding line in variable `maxline`. Once it has looped over all lines, it prints them out.

Did you catch that this one-liner doesn't work for lines with all negative numbers? Can you think of a solution?

Here is a solution that works for lines that have all negative numbers:

```
awk 'NR == 1 { max = $1; maxline = $0; next; }
$1 > max { max=$1; maxline=$0 };
END { print max, maxline }'
```

It works because it registers the maximum element on the first line when `NR == 1`. Then all the following lines get compared to what was on the first line, rather than to undefined, which happened in the first solution.

The `next` function makes Awk skip to the next line and not continue the execution of the script. It starts again with the first pattern in the script. Since it's the 2nd line now, pattern `NR == 1` is false and the pattern `$1 > max` is tested. The same is done for lines 3, 4, ..., etc.

3.12 Print the number of fields in each line, followed by the line

```
awk '{ print NF ":" $0 }'
```

This one-liner just prints out the predefined variable `NF` – Number of Fields, which contains the number of fields in the line, followed by a colon and the line itself. Very simple.

3.13 Print the last field of each line

```
awk '{ print $NF }'
```

Fields in Awk need not be referenced by constants. For example, code like `f = 3; print $f` would print out the 3rd field. This one-liner prints the `NF`-th field, which is the last field on the line.

3.14 Print the last field of the last line

```
awk '{ field = $NF }; END { print field }'
```

This one-liner always keeps the track of the last field in variable `field`. Once it has looped over all the lines, variable `field` contains the last field of the last line, and it just prints it out in the `END` block.

Here is a better version of the same one-liner. It's more common, idiomatic and efficient:

```
awk 'END { print $NF }'
```

However, it doesn't work in all Awks. It only works in POSIX awk, Gawk, Busybox awk, mawk and FreeBSD awk. It doesn't work in Solaris awk, tawk and jawk. I found this info in the Awk Feature Comparison Chart. Save this web page for later.

3.15 Print every line with more than 4 fields

```
awk 'NF > 4'
```

This one-liner omits the action statement. As I noted in one-liner 2.1, a missing action statement is equivalent to `{ print }`. So this one-liner is actually:

```
awk 'NF > 4 { print }'
```

It prints only those lines, which have `NF` larger than 4, that is, lines with more than 4 fields.

3.16 Print every line where the value of the last field is greater than 4

```
awk '$NF > 4'
```

This one-liner is similar to 3.13. It references the last field by the `NF` variable. If the value of the last field `$NR` is greater than 4, it prints it out.

Four

Text Conversion and Substitution

4.1 Convert Windows/DOS newlines (CRLF) to Unix newlines (LF) from Unix

```
awk '{ sub(/\r$/,""); print }'
```

This one-liner uses the `sub(regex, repl, [string])` function. This function substitutes the first instance of the regular expression `regex` in the string `string` with the string `repl`. If `string` is omitted, variable `$0` is used. Variable `$0`, as I explained in the very first one-liner contains the entire line, without the trailing `\n` character.

The one-liner replaces `\r` (CR char) character at the end of the line with nothing, i.e., erases CR at the end. The `$` at the end of regular expression makes sure only the trailing `\r` is matched. The `print` statement prints out the line and appends `ORS` variable, which is `\n` by default. Thus, a line ending with CRLF has been converted to a line ending with LF.

Also note that `sub` function does string replacement in-place. The return value is the number of elements substituted.

4.2 Convert Unix newlines (LF) to Windows/DOS newlines (CRLF) from Unix

```
awk '{ sub(/$/,"\r"); print }'
```

This one-liner also uses the `sub` function. This time it replaces the zero-width anchor `$` at the end of the line with a `\r` (CR). This substitution actually adds a CR character to the end of the line. After doing that Awk prints out the line and appends the `ORS`, making the line terminate with CRLF.

4.3 Convert Unix newlines (LF) to Windows/DOS newlines (CRLF) from Windows/DOS

```
awk 1
```

This one-liner may work, or it may not. It depends on the implementation. If the implementation catches the Unix newlines in the file, then it will read the file line by line correctly and output the lines terminated with CRLF. If it does not understand Unix LF's in the file, then it will print the whole file and terminate it with CRLF (single windows newline at the end of the whole file).

Statement `1` (or anything that evaluates to true) in Awk is syntactic sugar for `{ print }`.

4.4 Convert Windows/DOS newlines (CRLF) to Unix newlines (LF) from Windows/DOS

```
gawk -v BINMODE="w" '1'
```

Theoretically this one-liner should convert CRLFs to LFs on DOS. There is a note in GNU Awk documentation that says: "Under DOS, gawk (and many other text programs) silently translates end-of-line `\r\n` to `\n` on input and `\n` to `\r\n` on output. A special `BINMODE` variable allows control over these translations and is interpreted as follows: ... If `BINMODE` is `w`, then binary mode is set on write (i.e., no translations on writes)."

My tests revealed that no translation was done, so you can't really rely on this BINMODE hack.

It's better use the `tr` utility to convert CRLFs to LFs on Windows:

```
tr -d '\r'
```

The `tr` program is used for translating one set of characters to another. Specifying `-d` option makes it delete all characters and not do any translation. In this case it's the `\r` (CR) character that gets erased from the input. Thus, CRLFs become just LFs. Note that `\r` has to be escaped as `'\r'` so that the `tr` command receives two-character argument `\r`. If `'\r'` wasn't escaped, bash (and many other shells) would pass just single character `r` to `tr` and that wouldn't replace CR but would replace the literal character `r`.

4.5 Delete leading whitespace (spaces and tabs) from the beginning of each line (ltrim)

```
awk '{ sub(/^[ \t]+/, ""); print }'
```

This one-liner also uses `sub` function. What it does is replace regular expression `^[ \t]+` with nothing `""`. The regular expression `^[ \t]+` means – match one or more space or a tab at the beginning of the string. The `^` means beginning of the string and `[ \t]+` means match either a space or a tab one or more times.

4.6 Delete trailing whitespace (spaces and tabs) from the end of each line (rtrim)

```
awk '{ sub(/[ \t]+$/, ""); print }'
```

This one-liner is very similar to the previous one. It replaces regular expression `[ \t]+$` with nothing. The regular expression `[ \t]+$` means – match one or more space or a tab at the end of the string. The `+` in the regex means "match one or more".

4.7 Delete both leading and trailing whitespaces from each line (trim)

```
awk '{ gsub(/^[ \t]+|[ \t]+$/, ""); print }'
```

This one-liner uses a new function called `gsub`. The `gsub` function does the same as `sub`, except it performs as many substitutions as possible (that is, it's a global `sub`). For example, given a variable `f = "foo"`,

`sub("o", "x", f)` would replace just one "o" in variable `f` with "x", making `f` be "fxo"; but `gsub("o", "x", f)` would replace both "o"s in "foo" resulting "fxx".

The one-liner combines both previous one-liners 4.5 and 4.6 – it replaces leading whitespace `^[ \t]+` and trailing whitespace `[ \t]+$` with nothing, thus trimming the string. It uses the special alternation metacharacter `|` in the regular expression to match both leading and trailing whitespace characters.

To remove whitespace between fields you may use this one-liner:

```
awk '{ $1=$1; print }'
```

This is a pretty tricky one-liner. It seems to do nothing, right? Assign `$1` to `$1`. But no, when you change a field, Awk rebuilds the `$0` variable. It takes all the fields and concats them, separated by `OFS` – Output Field Separator, which is a single space by default. All the whitespace between the fields is gone.

4.8 Insert 5 blank spaces at beginning of each line

```
awk '{ sub(/^/, "     "); print }'
```

This one-liner substitutes the zero-length beginning of line anchor ^ with five empty spaces. As the anchor is zero-length and matches the beginning of line, the five whitespace characters get appended to beginning of the line.

4.9 Align all text to the right right on a 79-column width

```
awk '{ printf "%79s\n", $0 }'
```

This one-liner asks `printf` to print the string in `$0` variable and left pad it with spaces until the total length is 79 chars.

Please see the documentation of `printf` function for more information and examples. Documentation is available in `man 3 printf` man page.

4.10 Center all text on a 79-character width

```
awk '{ l=length(); s=int((79-l)/2);
printf "%"(s+l)"s\n", $0 }'
```

First this one-liner calculates the `length` of the line and puts the result in variable `l`. The function `length(var)` returns the string length of `var`. If the variable is not specified, it returns the length of the entire line (variable `$0`). Next it calculates how many white space characters to pad the line with and stores the result in variable `s`. Finally it `printf`s the line with the appropriate number of whitespace chars.

For example, when printing a string "foo", it first calculates the length of "foo" which is 3. Next it calculates the center column for 79-character width. That is, how many characters from the left the string "foo" should appear. It's easy to see that it's at (79-3)/2 = 38. Then it calculates `s+l` for the `printf` string. Since `%<number>s` right aligns the string, we have to take the length of the string `l` into account. So `s+l` is 41. Finally it calls `printf("%41s", "foo")`. The `printf` function outputs 38 spaces and then "foo" (41 chars total), making that string centered on a 79-character width.

4.11 Substitute (find and replace) "foo" with "bar" on each line

```
awk '{ sub(/foo/,"bar"); print }'
```

This one-liner is very similar to the others we have seen before. It uses the `sub` function to replace "foo" with "bar". It replaces any word that has "foo" in it, such as "bigfoo1700" or "foobaz". Please note that it replaces just the first match. To replace all "foo"s with "bar"s use the `gsub` function:

```
awk '{ gsub(/foo/,"bar"); print }'
```

Another way is to use the `gensub` function:

```
gawk '{ $0 = gensub(/foo/,"bar",4); print }'
```

This one-liner replaces only the 4th match of "foo" with "bar". It uses a never before seen `gensub` function. The prototype of this function is `gensub (regex, s, h[, t])`. It searches the string `t` for `regex` and replaces `h`-th match with `s`. If `t` is not given, `$0` is assumed. Unlike `sub` and `gsub` it returns the modified string `t` (remember that `sub` and `gsub` modified the string in-place).

Looking at `gensub(regex, s, h[, t])` prototype, we can see that in this one-liner the `regex` is `/foo/`, `s` is `"bar"`, `h` is `4`, and `t` is `$0`. It replaces the 4th instance of `"foo"` with `"bar"` and assigns the new string back to the whole line `$0`.

Please note that the `gensub` function is a non-standard function and requires GNU Awk or Awk included in NetBSD.

4.12 Substitute "foo" with "bar" only on lines that contain "baz"

```
awk '/baz/ { gsub(/foo/, "bar") }; { print }'
```

As I explained in the first one-liner, every Awk program consists of a sequence of pattern-action statements `pattern { action statements }`. Action statements are applied only to lines that match pattern.

In this one-liner the pattern is a regular expression `/baz/`. If line contains "baz", the action statement `gsub(/foo/, "bar")` is executed. And as we just learned, it substitutes all instances of "foo" with "bar". If you want to substitute just one, use the `sub` function!

4.13 Substitute "foo" with "bar" only on lines that don't contain "baz"

```
awk '!/baz/ { gsub(/foo/, "bar") }; { print }'
```

This one-liner negates the pattern `/baz/`. It works exactly the same way as the previous one, except it operates on lines that do not match this pattern.

4.14 Change "scarlet" or "ruby" or "puce" to "red"

```
awk '{ gsub(/scarlet|ruby|puce/, "red"); print}'
```

This one-liner makes use of extended regular expression alternation operator | (pipe). The regular expression `/scarlet|ruby|puce/` says: match "scarlet" or "ruby" or "puce". If the line matches, `gsub` replaces all the matches with "red".

4.15 Reverse order of lines (emulate "tac")

```
awk '{ a[i++] = $0 } END { for (j=i-1; j>=0;)
print a[j--] }'
```

This is the trickiest one-liner in this chapter. It starts by recording all the lines in the array `a`. For example, if the input to this program was three lines "foo", "bar", and "baz", then the array `a` would contain the following values: `a[0] = "foo"`, `a[1] = "bar"`, and `a[2] = "baz"`.

When the program has finished processing all lines, Awk executes the `END { }` block. The `END` block loops over the elements in the array `a` in reverse order and prints the recorded lines. In our example with "foo", "bar", "baz" the `END` block does the following:

```
for (j = 2; j >= 0; ) print a[j--]
```

First it prints out `j[2]`, then `j[1]` and then `j[0]`. The output is three separate lines "baz", "bar" and "foo". As you can see the input was reversed.

4.16 Join a line ending with a backslash with the next line

```
awk '/\\$/ { sub(/\\$/,""); getline t; print $0 t;
next }; 1'
```

This one-liner uses regular expression `/\\$/` to look for lines ending with a backslash. If the line ends with a backslash, the backslash gets removed by `sub(/\\$/,"")` function. Then the `getline t` function is executed. The `getline t` reads the next line from input and stores it in variable `t`. The `print $0 t` statement prints the original line (but with trailing backslash removed) and the newly read line (which was stored in variable `t`). Awk then continues with the `next` line and processing starts from the `/\\$/` pattern. If the line does not end with a backslash, Awk just prints it out with `1`.

4.17 Print and sort the login names of all users

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

I mentioned the `-F` argument in the introduction but didn't really explain it. Here is the full explanation of what it does. This argument specifies a character, a string or a regular expression that will be used to split the line into fields (`$1`, `$2`, ...). For example, if the line is "foo-bar-baz" and `-F` is "`-`", then the line will be split into three fields: `$1 = "foo"`, `$2 = "bar"`

and `$3 = "baz"`. If `-F` is not set to anything, the line will contain just one field `$1 = "foo-bar-baz"`.

Specifying `-F` is the same as setting the `FS` (Field Separator) variable in the `BEGIN` block of Awk program:

```
awk -F ":"
```

Is the same as:

```
awk 'BEGIN { FS=":" }'
```

`/etc/passwd` is a plain-text file, that contains a list of the system's accounts, along with some useful information like login name, user ID, group ID, home directory, shell, etc. The entries in the file are separated by a colon ":".

Here is an example of a line from `/etc/passwd` file:

```
pkrumins:x:1000:1700:Peteris Krumins:/home/pkrumins:/bin/bash
```

If we split this line on ":", the first field is the username (pkrumins in this example). The one-liner does just that – it splits the line on ":", then forks the `sort` program and feeds it all the usernames, one by one. After Awk has finished processing the input, sort program sorts the usernames and outputs them.

4.18 Print the first two fields in reverse order on each line

```
awk '{ print $2, $1 }' file
```

This one-liner is obvious. It reverses the order of fields `$1` and `$2`. For example, if the input line is "foo bar", then after running this program the output will be "bar foo".

4.19 Swap first field with second on every line

```
awk '{ temp = $1; $1 = $2; $2 = temp; print }'
```

This one-liner uses a temporary variable called `temp`. It assigns the first field `$1` to `temp`, then it assigns the second field to the first field and finally it assigns `temp` to `$2`. This procedure swaps the first two fields on every line. For example, if the input is "foo bar baz", then the output will be "bar foo baz".

4.20 Delete the second field on each line

```
awk '{ $2 = ""; print }'
```

This one-liner just assigns empty string to the second field. It's gone.

Remember that assigning to the fields makes Awk recompute the `$0` variable, which contains the entire line. In this one-liner we deleted the 2nd field by assigning nothing to it. When Awk recomputes the `$0`, it sees that `$2` is empty so there is nothing to include in `$0`.

4.21 Print the fields in reverse order on every line

```
awk '{ for (i=NF; i>0; i--) printf("%s ", $i);
printf("\n") }'
```

We saw the `NF` variable that stands for Number of Fields in Chapter 1. After processing each line, Awk sets the `NF` variable to number of fields found on that line.

This one-liner loops in reverse order starting from `NF` to `1` and outputs the fields one by one. It starts with field `$NF`, then `$(NF-1)`, ..., `$1`. After that it prints a newline character.

4.22 Remove duplicate, consecutive lines (emulate "uniq")

```
awk 'a != $0; { a = $0 }'
```

Variables in Awk don't need to be initialized or declared before they are being used. They come into existence the first time they are used. This one-liner uses variable `a` to keep the last line seen `{ a = $0 }`. Upon reading the next line, it compares if the previous line (in variable `a`) is not the same as the current one `a != $0`. If it is not the same, the expression evaluates to `1` (true), and as I explained earlier, any true expression is the same as `{ print }`, so the line gets printed out. Then the program saves the current line in variable `a` again and the same process continues over and over again.

4.23 Remove duplicate, nonconsecutive lines

```
awk '!a[$0]++'
```

This one-liner is very idiomatic. It registers the lines seen in the associative-array `a` (arrays are always associative in Awk) and at the same time tests if it has seen the line before. If it has seen the line before, then `a[line] > 0` and `!a[line]` is `0`. Any expression that evaluates to false is a no-op, and any expression that evals to true is equal to `{ print }`. So if it has seen a line before `!a[line]` is a no-op, but if the line hasn't been seen then it gets printed out.

For example, suppose the input is:

```
foo
bar
foo
baz
```

When Awk sees the first "foo", it evaluates the expression `!a["foo"]++`. `a["foo"]` is false, but `!a["foo"]` is true, so Awk prints out "foo". Then it increments `a["foo"]` by one with `++` post-increment operator. Array `a` now contains one value `a["foo"] == 1`.

Next Awk sees "bar", it does exactly the same what it did to "foo" and prints out "bar". Array "a" now contains two values `a["foo"] == 1` and `a["bar"] == 1`.

Now Awk sees the second "foo". This time `a["foo"]` is true (it's `1`), `!a["foo"]` is false and Awk does not print anything! Array `a` still contains two values `a["foo"] == 2` and `a["bar"] == 1`.

Finally Awk sees "baz" and prints it out because `!a["baz"]` is true. Array `a` now contains three values `a["foo"] == 2` and `a["bar"] == 1` and `a["baz"] == 1`.

The output is:

```
foo
bar
baz
```

Here is another one-liner to do the same:

```
awk '!($0 in a) { a[$0]; print }'
```

It's basically the same as the one above, except that it uses the `in` operator. Given an array `a`, an expression `foo in a` tests if the value of variable `foo` is in `a`.

Note that the empty statement `a[$0]` creates an element in the array.

4.24 Concatenate every 5 lines of input with a comma

```
awk 'ORS=NR%5?", ":"\n"'
```

We saw the `ORS` variable in the first one-liner. This variable gets appended after every line that gets output. In this one-liner it gets changed on every 5th line from a comma to a newline. For lines 1, 2, 3, 4 it's a comma, for line 5 it's a newline, for lines 6, 7, 8, 9 it's a comma, for line 10 a newline, etc.

Here is an example. Suppose the input is this:

```
line 1
line 2
line 3
line 4
line 5
line 6
line 7
line 8
line 9
line 10
```

Then the output is:

```
line 1, line 2, line 3, line 4, line 5
line 6, line 7, line 8, line 9, line 10
```

The statement `NR % 5 ? ", " : "\n"` uses the ternary operator `test ? true : false`, which evaluates the `test` expression and returns `true` if `test` was true, and returns `false` otherwise. The `NR` variable is the current line number, so it goes like 1, 2, 3, ..., etc. The operation `NR % 5` means the remainder of `NR` divided by 5. If the remainder is not 0 (value is true), the ternary operator returns comma `", "`, otherwise it returns newline `"\n"`. So at every 5th line the value `NR % 5` is 0 (false) and `ORS` gets set to the newline, and for all other lines it's a comma. Simple, isn't it?

Five

Selective Printing and Deleting of Certain Lines

5.1 Print the first 10 lines of a file (emulates "head -10")

```
awk 'NR < 11'
```

Awk has a special variable called `NR` that stands for "Number of Lines seen so far in the current file". After reading each line, Awk increments this variable by one. So for the first line it's 1, for the second line 2, ..., etc. As I explained in the very first one-liner, every Awk program consists of a sequence of pattern-action statements `pattern { action statement }`. The `action statements` part get executed only on those lines that match the `pattern` (pattern evaluates to true). In this one-liner the pattern is `NR < 11` and there is no `action statement`. The default action in case of a missing `action statement` is to print the line as-is (it's equivalent to `{ print $0 }`). The pattern in this one-liner is an expression that tests if the current line number is less than 11. If the line number is less than 11, Awk prints the line. As soon as the line number is 11 or more, the pattern evaluates to false and Awk skips the line.

A much better way to do the same is to quit after seeing the first 10 lines (otherwise we are looping over all lines above 10 and doing nothing):

```
awk '1; NR == 10 { exit }'
```

The `NR == 10 { exit }` part guarantees that as soon as the line number 10 is printed, Awk quits. For line numbers less than or equal to 10, Awk evaluates `1` that is always a true-statement. And as we learned earlier, true statements without the "action statement" part are equal to `{ print $0 }`.

5.2 Print the first line of a file (emulates "head -1")

```
awk 'NR > 1 { exit }; 1'
```

This one-liner is very similar to the previous one. The `NR > 1` is true only for lines greater than one, so it does not get executed on the first line. On the first line only the `1`, the true statement, gets executed. It makes Awk print the line and read the next line. Now the `NR` variable is 2, and `NR > 1` is true. At this moment `{ exit }` gets executed and Awk quits. That's it. Awk printed just the first line of the file.

Another way to do the same:

```
awk 'NR == 1 { print; exit }'
```

Here we test if the current line is the first one, if it is, we print it and exit. Done!

5.3 Print the last 2 lines of a file (emulates "tail -2")

```
awk '{ y=x "\n" $0; x=$0 }; END { print y }'
```

Okay, so what does this one do? First of all, notice that `{ y=x "\n" $0; x=$0 }` action statement group is missing the pattern. When the pattern is missing, Awk executes the statement group for all lines. At the first line, it sets variable `y` to `\nline1` (because `x` is not yet defined). At the second line it sets variable `y` to `line1\nline2`. At the third line it sets variable `y` to `line2\nline3`. As you can see, at line N it sets the variable `y` to `lineN-1\nlineN`. Finally, when it reaches EOF, variable `y` contains the last two lines and they get printed via `print y` statement at the `END` block.

Thinking about this one-liner for a second we can conclude that it is very ineffective – it reads the whole file line by line just to print out the last two lines! Unfortunately there is no `seek` statement in Awk, so you can't

seek to the end-2 lines in the file (that's what `tail` does). I recommend using `tail -2` to print the last 2 lines of a file.

5.4 Print the last line of a file (emulates "tail -1")

```
awk 'END { print }'
```

This one-liner may or may not work. It relies on an assumption that the `$0` variable that contains the entire line does not get reset after all the input has been exhausted. The special `END` pattern gets executed after the input has been exhausted (or `exit` is called). In this one-liner the `print` statement is supposed to print `$0` at EOF, which may or may not have been reset.

If it will work depends on your `awk` program's version and implementation. It works with GNU Awk for example, but doesn't seem to work with nawk or xpg4/bin/awk.

The most compatible way to print the last line is:

```
awk '{ rec=$0 } END{ print rec }'
```

Just like the previous one-liner, it's computationally expensive to print the last line of the file this way, and `tail -1` should be the preferred way.

5.5 Print only the lines that match a regular expression "/regex/" (emulates "grep")

```
awk '/regex/'
```

This one-liner uses a regular expression `/regex/` as a pattern. If the current line matches the regex, it evaluates to true, and Awk prints the line (remember that missing action statement is equal to `{ print }` that prints the whole line).

5.6 Print only the lines that do not match a regular expression "/regex/" (emulates "grep -v")

```
awk '!/regex/'
```

Pattern matching expressions can be negated by appending `!` in front of them. If they were to evaluate to true, appending `!` in front makes them evaluate to false, and the other way around. This one-liner inverts the regex match of the previous one-liner and prints all the lines that do not match the regular expression `/regex/`.

5.7 Print the line immediately before a line that matches "/regex/" (but not the line that matches itself)

```
awk '/regex/ { print x }; { x=$0 }'
```

This one-liner always saves the current line in the variable `x`. When it reads in the next line, the previous line is still available in the `x` variable. If this line matches `/regex/`, it prints out the variable `x`, and as a result, the previous line gets printed.

It does not work, if the first line of the file matches `/regex/`, in that case, we might want to print "match on line 1". Here is how to do it:

```
awk '/regex/ { print (NR==1 ? "match on line 1" : x) };
{ x=$0 }'
```

In case the current line matches `/regex/`, this one-liner tests if the current line is the first. If it is, it "match on line 1" gets printed. Otherwise variable `x` gets printed (that as we found out in the previous one-liner contains the previous line). Notice that this one-liner uses a ternary operator `foo?bar:baz` that is short for "if foo, then bar, else baz". I explained the ternary operator in more details in one-liner 4.24.

5.8 Print the line immediately after a line that matches "/regex/" (but not the line that matches itself)

```
awk '/regex/ { getline; print }'
```

This one-liner calls the `getline` function on all the lines that match `/regex/`. This function sets `$0` to the next line (and also updates `NF`, `NR`, `FNR` variables). The `print` statement then prints this next line. As a result, only the line after a line matching `/regex/` gets printed.

If it is the last line that matches `/regex/`, then `getline` actually returns error and does not set `$0`. In this case the last line gets printed itself.

5.9 Print lines that match any of "AAA" or "BBB", or "CCC"

```
awk '/AAA|BBB|CCC/'
```

This one-liner uses a feature of extended regular expressions that support the | or alternation meta-character. This meta-character separates "AAA" from "BBB", and from "CCC", and tries to match them separately on each line. Only the lines that contain one (or more) of them get matched and printed.

5.10 Print lines that contain "AAA", "BBB", and "CCC" in this order

```
awk '/AAA.*BBB.*CCC/'
```

This one-liner uses a regular expression "AAA.*BBB.*CCC" to print lines. This regular expression says, "match lines containing AAA followed

by any text, followed by BBB, followed by any text, followed by CCC in this order". If a line matches, it gets printed.

For example, the line "AAA1711BBB777CCC" matches but "BBB1711AAA777CCC" doesn't.

5.11 Print only the lines that are 65 characters in length or longer

```
awk 'length > 64'
```

This one-liner uses the `length` function. This function is defined as `length([str])` – it returns the length of the string `str`. If none is given, it returns the length of the string in variable `$0`. For historical reasons, parenthesis `()` at the end of `length` can be omitted. This one-liner tests if the current line is longer than 64 chars, if it is, the `length > 64` evaluates to true and line gets printed.

5.12 Print only the lines that are less than 64 characters in length

```
awk 'length < 64'
```

This one-liner is almost byte-by-byte equivalent to the previous one. Here it tests if the length of the line is less than 64 characters. If it is, Awk prints it out. Otherwise nothing gets printed.

5.13 Print a section of file from regular expression to end of file

```
awk '/regex/,0'
```

This one-liner uses a pattern match in form `pattern1, pattern2` that is called the "range pattern". It matches all the lines starting with a line that matches `pattern1` and continues until a line matches `pattern2` (inclusive). In this one-liner `pattern1` is the regular expression `/regex/` and `pattern2` is just 0 (false). So this one-liner prints all lines starting from a line that matches `/regex/` continuing to end-of-file because 0 is always false, and `pattern2` never matches.

5.14 Print lines 8 to 12 (inclusive)

```
awk 'NR==8,NR==12'
```

This one-liner also uses a range pattern in format `pattern1, pattern2`. The `pattern1` here is `NR==8` and `pattern2` is `NR==12`. The first pattern means "the current line is 8th" and the second pattern means "the current line is 12th". This one-liner prints lines between these two patterns (inclusive).

5.15 Print line number 52

```
awk 'NR==52'
```

This one-liner tests to see if current line is number 52. If it is, `NR==52` evaluates to true and the line gets implicitly printed out (patterns without statements print the line unmodified).

The correct way, though, is to quit after line 52:

```
awk 'NR==52 { print; exit }'
```

This one-liner forces Awk to quit after line number 52 is printed. It is the correct way to print line 52 because there is nothing else to be done, so why loop over the whole file doing nothing.

5.16 Print section of a file between two regular expressions (inclusive)

```
awk '/Iowa/,/Montana/'
```

I explained what a range pattern such as `pattern1,pattern2` does in general in one-liner 5.13. In this one-liner `pattern1` is `/Iowa/` and `pattern2` is `/Montana/`. Both of these patterns are regular expressions. This one-liner prints all the lines starting with a line that matches "Iowa" and ending with a line that matches "Montana", inclusive.

5.17 Print all lines where 5th field is equal to "abc123"

```
awk '$5 == "abc123"'
```

This one-liner uses idiomatic Awk – if the given expression is true, Awk prints out the line. The fifth field is referenced by `$5` and it's checked to be equal to the string `abc123`. If it is, the expression is true and the line gets printed.

Unwinding this idiom, this one-liner is really equal to:

```
awk '{ if ($5 == "abc123") { print $0 } }'
```

But it's always better to be short and idiomatic.

5.18 Print any line where field #5 is not equal to "abc123"

```
awk '$5 != "abc123"'
```

This is exactly the same as previous one-liner, except it negates the comparison. If the fifth field `$5` is not equal to `abc123`, then print it.

Unwinding it, it's equal to:

```
awk '{ if ($5 != "abc123") { print $0 } }'
```

Another way is to literally negate the whole previous one-liner:

```
awk '!($5 == "abc123")'
```

5.19 Print all lines whose 7th field matches a regular expression

```
awk '$7 ~ /^[a-f]/'
```

This is also idiomatic Awk. It uses the ~ operator to test if the seventh `$7` field matches a regular expression `^[a-f]`. This regular expression means "all lines that start with a lower-case letter a, b, c, d, e, or f".

5.20 Print all lines whose 7th field doesn't match a regular expression

```
awk '$7 !~ /^[a-f]/'
```

This one-liner negates the previous one-liner and prints all lines that do not start with a lower-case letter a, b, c, d, e, and f. The opposite of match operator ~ is `!~`.

Another way to write the same is:

```
awk '$7 ~ /^[^a-f]/'
```

Here we negated the group of letters `[a-f]` by adding `^` in the group. That's a regex trick to know.

5.21 Delete all blank lines from a file

```
awk NF
```

This one-liner uses the special `NF` variable that contains number of fields on the line. For empty lines, `NF` is 0, that evaluates to false, and false statements do not get the line printed.

Another way to do the same is:

```
awk '/./'
```

This one-liner uses a regular-expression `/./` that matches any single character. Empty lines do not have any characters, so it does not match.

Six

String and Array Creation

6.1 Create a string of a specific length (generate a string of x's of length 513)

```
awk 'BEGIN { while (a++<513) s=s "x"; print s }'
```

This one-liner uses the `BEGIN { }` special block that gets executed before anything else in an Awk program. In this block a while loop appends the character `x` to variable `s` 513 times. After it has looped, the `s` variable gets printed out. As this Awk program does not have a body, it quits after executing the BEGIN block.

This one-liner printed the 513 `x`'s out, but you could have used these `x`'s for anything you wish in `BEGIN`, main program or `END` blocks.

Unfortunately this is not the most effective way to do it. It's a quadratic time solution because the string grows as "s", "ss", "sss", ... So if we sum the work required to create the string we get the result 1 + 2 + 3 + ... + n, which is $\frac{n(n+1)}{2} = O(n^2)$ – quadratic time solution.

Here is a solution that's linear time:

```
function rep(str, num,     remain, result) {
    if (num < 2) {
        remain = (num == 1)
    } else {
        remain = (num % 2 == 1)
        result = rep(str, (num - remain) / 2)
    }
    return result result (remain ? str  : "")
}
```

This function can be used as following:

```
awk 'BEGIN { s = rep("x", 513) }'
```

6.2 Insert a string of specific length at a certain character position (insert 49 x's after 6th char)

```
gawk --re-interval 'BEGIN{ while(a++<49) s=s "x" };
{ sub(/^.{6}/,"&" s) }; 1'
```

This one-liner works only with Gnu Awk and several others, because it uses the interval expression `.{6}` in the Awk program's body. Interval expressions were not traditionally available in awk, that's why you have to use `--re-interval` option to enable them.

For those that do not know what interval expressions are, they are regular expressions that match a certain number of characters. For example, `.{6}` matches any six characters (the any char is specified by the dot `.`). An interval expression `b{2,4}` matches at least two, but not more than four `b` characters. To match words, you have to give them higher precedence – `(foo){4}` matches `foo` repeated four times – `foofoofoofoo`.

The one-liner starts the same way as the previous – it creates a 49 character string `s` in the BEGIN block. Next, for each line of the input, it calls the `sub` function that replaces the first 6 characters with themselves and `s` appended. The `&` in the `sub` function means the matched part of regular expression. The `"&" s` means matched part of regex and contents of variable `s`. The `1` at the end of whole Awk one-liner prints out the modified line (it's syntactic sugar for just `print` (that itself is syntactic sugar for `print $0`)).

The same can be achieved with normal standard Awk:

```
awk 'BEGIN{ while(a++<49) s=s "x" };
{ sub(/^....../,"&" s) }; 1
```

Here we just match six chars `......` at the beginning of line, and replace them with themselves + contents of variable `s`.

It may get troublesome to insert a string at 29th position for example. You'd have to go tapping "." twenty-nine times `.....` Better use Gnu Awk then and write `.{29}`.

Actually, if you look at the Awk Feature Comparison Chart, you can see that this solution with `.{6}` would also work with POSIX awk, Busybox awk, and Solaris awk.

6.3 Create an array from string

```
split("Jan Feb Mar Apr May Jun Jul Aug Sep Oct
Nov Dec", month, " ")
```

This is not a one-liner per se but a technique to create an array from a string. This technique can also be used in other languages, such as, Python and JavaScript. We use the `split(str, arr, regex)` function to do that. It splits string `str` into fields by regular expression `regex` and puts the fields in array `arr`. The fields are placed in `arr[1]`, `arr[2]`, ..., `arr[N]`. The `split` function itself returns the number of fields the string was split into.

In this piece of code the regex is simply space character " ", the array is month and string is "Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec". After the split, `month[1]` is "Jan", `month[2]` is "Feb", ..., `month[12]` is "Dec".

6.4 Create an array named "mdigit", indexed by strings

```
for (i=1; i<=12; i++) mdigit[month[i]] = i
```

This is another array creation technique and not a real one-liner. This technique creates a reverse lookup array. Remember from the previous one-liner that `month[1]` was "Jan", ..., `month[12]` was "Dec". Now we want to do the reverse lookup and find the number for each month. To do that

we create a reverse lookup array `mdigit`, such that `mdigit["Jan"] = 1`, ..., `mdigit["Dec"] = 12`.

It's really trivial, we loop over `month[1]`, `month[2]`, ..., `month[12]` and set `mdigit[month[i]]` to i. This way `mdigit["Jan"] = 1` etc.

A

Awk Special Variables

In this appendix I'll summarize the most commonly used special variables, such as `FS`, `OFS`, `NR`, `NR`, and others.

A.1 FS – Input Field Separator

The `FS` stands for Input **F**ield **S**eparator. It is used by Awk to split the line into fields for quick access via `$1`, `$2`, ..., variables. By default `FS` is one or more spaces.

For example, if the input line is:

```
whats-going-on-here
```

And the `FS` is set to `-`, then quick access variables will be:

```
$1 = whats
$2 = going
$3 = on
$4 = here
```

You can set the `FS` variable in three ways:

1. Use the special `-F` command line option.
   For example, to set `FS` to colon:

   ```
   awk -F: '{print $1}'
   ```

2. Set the `FS` variable in the `BEGIN` block.
   For example,

```
awk 'BEGIN { FS=":" } {print $1}'
```

3. Set the `FS` variable with `-v var=val` command line option.
   For example,

```
awk -v FS=: '{print $1}'
```

A.2 OFS – Output Field Separator

The `OFS` stands for **O**utput **F**ield **S**eparator. It is used by Awk to separate fields `$1`, `$2`, ..., when they get printed. `OFS` is space by default.

Here is an example, suppose the input is:

```
awk is awesome
```

And the Awk program sets `OFS` to `:` and prints the first and third fields:

```
awk -v OFS=: '{ print $1, $3 }'
```

Then the output is going to be:

```
awk:awesome
```

Notice the comma between `$1` and `$3` in the `print` statement. If there was no comma, then the fields would have gotten concatenated and the output would had been "awkawesome".

Here is a trick Awk one-liner. If you modify any of the fields, the whole line gets recomputed in the `$0` variable and the fields in the line get separated by `OFS`. For example, if you have a line where the fields are separated by spaces and you wish to separate them by a hyphen, do this:

```
awk -v OFS=- '{ $1=$1; print }'
```

The trick here is to assign field `$1` to itself. This causes the entire line in `$0` to be recomputed. Since we set `OFS` to `-`, the fields get separated by a hyphen.

To explain it even better, check this example out. Suppose the input is:

```
Awk is a fun language
```

And the Awk program is the same as above:

```
awk -v OFS=- '{ $1=$1; print }'
```

Then the output will be:

```
Awk-is-a-fun-language
```

A.3 NF – Number of Fields on the current line

The `NF` stands for **N**umber of **F**ields on the current line after the line has been split by `FS` field separator. For example, if the line is:

```
today is a sunny day
```

And you have the simplest possible Awk program:

```
awk '{ print NF }'
```

Then the output is going to be 5, because there are 5 fields in the line.

However, if you change `FS` to `-`:

```
awk -F- '{ print NF }'
```

Then the output is going to be 1, because there is just one field, which is the entire line.

A.4 NR – Number of records seen so far (current line number)

The `NR` stands for the **N**umber of **R**ecords seen so far or simply the current line number. There is not much more to it. When Awk reads the first line (record), `NR` is 1, when it reads the next line, `NR` is 2, etc.

To give an example, this one-liner numbers all the lines:

```
awk '{ print NR ". " $0 }'
```

I explained this one-liner in more details in one-liner 3.2.

A.5 RS – Input Record Separator

The `RS` stands for Input **R**ecord **S**eparator. It's a newline by default and it's used by Awk to split the input into records. As it's a newline by default, I am used to thinking of lines and records as the same thing but it's not. If you set the `RS` to, let's say, space, then suddenly every piece of text that is separated by space is a record and gets put in the `$0` variable for processing.

For example, let's say the input is:

```
foo bar baz:one two three
```

And the Awk program is:

```
awk 'BEGIN { RS=":" } { print }'
```

Then the output is going to be:

```
foo bar baz
one two three
```

That's because instead of a newline, the record separator now was a colon, so Awk found two records and `print`ed them out.

A.6 ORS – Output Record Separator

The `ORS` stands for **O**utput **R**ecord **S**eparator. The output record separator gets output after each record (line). It's a newline by default.

Here is an example. Suppose the input is:

```
line 1
line 2
line 3
```

And the Awk program is:

```
awk -v ORS=" X " '{ print }'
```

Then the output is going to be:

```
line 1 X line 2 X line 3 X
```

The X at the end is ugly, to get rid of it you can pipe the output of Awk to sed:

```
awk -v ORS=" X " '{ print }' | sed 's/ X $//'
```

The output then is:

```
line 1 X line 2 X line 3
```

The sed expression `s/ X $//` means replace the trailing " X " with nothing. Actually, talking about sed, my next book is going to be Sed One-Liners Explained (http://catonmat.net/blog/sed-book). I'll announce it on my blog when it's done!

B

Idiomatic Awk

Very often in the book I mention the term "Idiomatic Awk". Let me explain what I mean by that.

Every language, including Awk, has idioms. Idioms are the most elegant way to write something. Let me explain that by an example.

Suppose you want to print all lines that have 5 fields. One obvious way to write it is the following:

```
awk '{ if (NF == 5) { print $0 } }'
```

This is good and it works but we can start to apply idioms to simplify this example. Here is the first step – when writing `print $0` we don't really need to write `$0` out. Calling just `print` is the same as calling `print $0`. So we can rewrite this one-liner this way:

```
awk '{ if (NF == 5) { print } }'
```

Next we can remember that all Awk programs are of form `pattern { action }` and if the `pattern` evaluates to true, the action is applied to the line. In our example we're explicitly testing if the line has 5 fields, but we can bring this test into the pattern and write it the following way:

```
awk 'NF == 5 { print }'
```

This is starting to look very idiomatic now but we're not yet done. As I explained in the first one-liner, only if the pattern is true, the action gets executed, and if the action is missing, it's equivalent to `{ print }`. Aha! We can get rid of that print statement and make the one-liner look as idiomatic as it can be:

```
awk 'NF == 5'
```

Isn't this awesome? We were to remove the `if` statement and a bunch of curly braces and make the program much shorter. I love it when I am able to write idiomatic code!

Index