Eric Pement's Awk one-liner

* 1. [File spacing](http://www.catonmat.net/blog/awk-one-liners-explained-part-one/) (explained in part one).
* 2. [Numbering and calculations](http://www.catonmat.net/blog/awk-one-liners-explained-part-one/) (explained in part one).
* 3. [Text conversion and substitution](http://www.catonmat.net/blog/awk-one-liners-explained-part-two/) (explained in part two).
* 4. [Selective printing of certain lines](http://www.catonmat.net/blog/awk-one-liners-explained-part-three/) (explained in part three).
* 5. [Selective deletion of certain lines](http://www.catonmat.net/blog/awk-one-liners-explained-part-three/) (explained in part three).
* 6. **String creation, array creation and update on selective printing of certain lines**. (explained in this part).
* 7. [Release of Awk One-Liners Explained e-book.](http://www.catonmat.net/blog/awk-book/)

**1. Line Spacing**

**1. Double-space a file.**

awk '1; { print "" }'

So how does it work? A one-liner is an Awk program and every Awk program consists of a sequence of pattern-action statements "**pattern { action statements }**". 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 }'. Thus, this one-liner translates to:

awk '1 { print } { print "" }'

An action is applied 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 an **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 a variable holding the entire line. The second print statement 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.

**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 previously, statement "1" gets translated to "{ print }", and every print statement gets terminated with the value of ORS variable.

**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 the current line was split into. For example, a line "this is a test" splits in four pieces and NF gets set to 4. The empty line "" does not split into any pieces and NF gets set to 0. Using NF as a pattern can effectively filter out empty lines. This one liner says: "If there are any number of fields, print the whole line followed by newline."

**4. Triple-space a file.**

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

This one-liner is very similar to previous ones. '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.

**2. Numbering and Calculations**

**5. 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. 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 one for the second file and number lines in this file from 1 to 12. FNR gets reset from file to file.

**6. Number lines for all files together.**

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

This one works the same as #5 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).

**7. Number lines in a fancy manner.**

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

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

**8. 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 variable 'a' each time the line is non-empty, then it appends the value of this variable to the beginning of line and prints it out.

**9. Count lines in files (emulates wc -l).**

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 (= number of lines in the file).

**10. 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 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.

**11. 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 #10, 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". It is necessary if 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. prints just the ORS). Adding a 0 does a mathematical operation and undef+0 = 0, so it prints "0".

**12. 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 (...) { ... }** 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 field number 5 to 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

}'

**13. 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, special pattern 'END { ... }' is executed, which prints the total number of fields. See 11th one-liner for explanation of why we "print total+0" in the END block.

**14. 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 "theBeth333"). 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.

**15. 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 (in variable 'max') and the corresponding line (in variable 'maxline'). Once it has looped over all lines, it prints them out. Warning: this one-liner does not work if all the values are negative.

Here is the fix:

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

**16. 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.

**17. 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 field with the value of NF. $NF is last field in the line.

**18. Print the last field of the last line.**

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

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

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

awk 'END { print $NF }'

**19. Print every line with more than 4 fields.**

awk 'NF > 4'

This one-liner omits the action statement. As I noted in one-liner #1, a missing action statement is equivalent to '{ print }'.

**20. Print every line where the value of the last field is greater than 4.**

awk '$NF > 4'

This one-liner is similar to #17. It references the last field by NF variable. If it's greater than 4, it prints it out.

## 3. Text Conversion and Substitution

**21. 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 regular expression "regex" in string "string" with the string "repl". If "string" is omitted, variable $0 is used. Variable $0, as I explained in [the first part of the article](http://www.catonmat.net/blog/awk-one-liners-explained-part-one/), contains the entire line.

The one-liner replaces '\r' (CR) character at the end of the line with nothing, i.e., erases CR at the end. 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.

**22. 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 char). 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.

**23. 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).

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

**24. 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](http://www.gnu.org/manual/gawk/html_node/PC-Using.html) 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 rely on this BINMODE hack.

Eric suggests to 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.

**25. 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 "\t" at the beginning "^" of the string.

**26. 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 "\t" at the end "$" of the string. The "+" means "one or more".

**27. Delete both leading and trailing whitespaces from each line (trim).**

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

This one-liner uses a new function called "gsub". Gsub() 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 - it replaces leading whitespace "^[ \t]+" and trailing whitespace "[ \t]+$" with nothing, thus trimming the string.

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 (single space by default). All the whitespace between fields is gone.

**28. 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.

**29. Align all text flush 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](http://www.gnu.org/manual/gawk/html_node/Basic-Printf.html) for more information and examples.

**30. 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". Length(var) function 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 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 column "foo" should appear which (79-3)/2 = 38. Finally it printf("%41", "foo"). Printf() function outputs 38 spaces and then "foo", making that string centered (38\*2 + 3 = 79)

**31. 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". 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" (sub and gsub modified the string in-place).

Gensub() is a non-standard function and requires GNU Awk or Awk included in NetBSD.

In this one-liner regex = "/foo/", s = "bar", h = 4, and t = $0. It replaces the 4th instance of "foo" with "bar" and assigns the new string back to the whole line $0.

**32. Substitute "foo" with "bar" only on lines that contain "baz".**

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

As I explained in the first one-liner in the first part of the article, 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 have learned, it substitutes all instances of "foo" with "bar". If you want to substitute just one, use the sub() function!

**33. Substitute "foo" with "bar" only on lines that do not 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 contain match this pattern.

**34. 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".

**35. 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 today. 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" 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.

**36. 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. "Getline t" reads the next line from input and stores it in variable t. "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. If the line does not end with a backslash, Awk just prints it out with "1".

Unfortunately this one liner fails to join more than 2 lines (this is left as an exercise to the reader to come up with a one-liner that joins arbitrary number of lines that end with backslash :)).

**37. Print and sort the login names of all users.**

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

This is the first time we see the -F argument passed to Awk. 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 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:1000:100: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.

**38. 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".

**39. 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".

Ps. This one-liner was incorrect in Eric's awk1line.txt file. "Print" was missing.

**40. Delete the second field on each line.**

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

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

**41. 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 the [part one](http://www.catonmat.net/blog/awk-one-liners-explained-part-one/) of this article. 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.

**42. 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.

This one-liner is actually incorrect. It uses a regular expression matching operator "!~". If the previous line was something like "fooz" and the new one is "foo", then it won't get output, even though they are not duplicate lines.

Here is the correct, fixed, one-liner:

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

It compares lines line-wise and not as a regular expression.

**43. 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 had seen the line before. If it had seen the line before, then a[line] > 0 and !a[line] == 0. Any expression that evaluates to false is a no-op, and any expression that evals to true is equal to "{ print }".

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 - 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, "!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:

foo

bar

baz

Here is another one-liner to do the same. Eric in his one-liners says it's the most efficient way to do it.

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

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

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

**44. Concatenate every 5 lines of input with a comma.**

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

We saw the ORS variable in [part one](http://www.catonmat.net/blog/awk-one-liners-explained-part-one/) of the article. 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.

## 4. Selective Printing of Certain Lines

**45. 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](http://www.catonmat.net/blog/awk-one-liners-explained-part-one/), every Awk program consists of a sequence of pattern-action statements "pattern { action statements }". The "action statements" part get executed only on those lines that match "pattern" (pattern evaluates to true). In this one-liner the pattern is "NR < 11" and there are no "action statements". The default action in case of missing "action statements" 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 lines > 10 and doing nothing):

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

The "NR == 10 { exit }" part guarantees that as soon as the line number 10 is reached, Awk quits. For lines smaller than 10, Awk evaluates "1" that is always a true-statement. And as we just learned, true statements without the "action statements" part are equal to "{ print $0 }" that just prints the first ten lines!

**46. Print the first line of a file (emulates "head -1").**

awk 'NR > 1 { exit }; 1'

This one-liner is very similar to 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.

**47. 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. For the first line, it sets variable "y" to "\nline1" (because x is not yet defined). For the second line it sets variable "y" to "line1\nline2". For the third line it sets variable "y" to "line2\nline3". As you can see, for 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.

Thinking about this one-liner for a second one concludes 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). It's recommended to use "tail -2" to print the last 2 lines of a file.

**48. 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 the input has been exhausted. The special "END" pattern gets executed after the input has been exhausted (or "exit" called). In this one-liner the "print" statement is supposed to print "$0" at EOF, which may or may not have been reset.

It depends on your awk program's version and implementation, if it will work. 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.

**49. 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).

**50. 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 (#49) one-liner and prints all the lines that do not match the regular expression "/regex/".

**51. 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 that 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", for example:

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

This one-liner tests if variable "x" contains something. The only time that x is empty is at very first line. In that case "match on line 1" gets printed. Otherwise variable "x" gets printed (that as we found out 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".

**52. 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.

**53. 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.

**54. Print lines that contain "AAA" and "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.

**55. 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.

**56. 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 if line less than 64 characters. If it is, Awk prints it out. Otherwise nothing gets printed.

**57. 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 "range pattern". The [3rd Awk Tip](http://www.catonmat.net/blog/ten-awk-tips-tricks-and-pitfalls/#awk_ranges) from article "[10 Awk Tips, Tricks and Pitfalls](http://www.catonmat.net/blog/ten-awk-tips-tricks-and-pitfalls/)" explains this match very carefully. It matches all the lines starting with a line that matches "pattern1" and continuing until a line matches "pattern2" (inclusive). In this one-liner "pattern1" is a 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).

**58. 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.

**59. 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 doing nothing.

**60. 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 #57. 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. Selective Deletion of Certain Lines

There is just one one-liner in this section.

**61. 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 match "." that matches any character. Empty lines do not have any characters, so it does not match.

## String Creation

**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 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 it 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 linear time solution. My friend **waldner** (who, by the way, wrote a guest post on [10 Awk Tips, Tricks and Pitfalls](http://www.catonmat.net/blog/ten-awk-tips-tricks-and-pitfalls/)) showed me a [solution](http://awk.freeshell.org/RepeatAString) that's logarithmic time (based on idea of [recursive squaring](http://www.catonmat.net/blog/mit-introduction-to-algorithms-part-two/)):

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) }'

**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, 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 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}".

Once again, my friend waldner corrected me and pointed to [Awk Feature Comparsion](http://awk.freeshell.org/AwkFeatureComparison) chart. The chart suggests that the original one-liner with ".{6}" would also work with POSIX awk, Busybox awk, and Solaris awk.

## Array Creation

**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. The split(Str, Arr, Regex) function is used 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".

**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 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.

## Selective Printing of Certain Lines

**5. 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 "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 } }'

**6. 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")'

**7. Print all lines whose 7th field matches a regular expression.**

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

This is also idiomatic Awk. It uses "~" 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".

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

This one-liner matches negates the previous one and prints all lines that do not start with a lower-case letter a, b, c, d, e, and f.

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.