# AWK and PERL and SED and REGEX

# AWK syntax

|  |
| --- |
| awk 'condition{action}' file or  awk 'condition{action}condition{action}condition{action}' file  where condition can be any of:   1. a key word like BEGIN or END 2. an arithmetic expression like var < 7 or NF or 1 3. a regexp comparison like $0 ~ "foo" or $0 ~ /foo/ or /foo/ or $0 ~ var or match($0,/foo/) 4. a string comparison like $0 == "foo" or index($0,"foo") 5. nothing at all in which case it's assumed to be true when there's an associated action block.   and probably other things I'm forgetting to list.  Eg:  $ awk '/Linux/{x=NR+2}(NR<=x){print}' file  Linux  Solaris  Aix   1. Condition{action} = /Linux/{x=NR+2} 2. Condition{action} = (NR<=x){print}   C-like syntax would be:  NR=0  x=0  While read (file,line)  {  NR++  -- First condition --  if (line ~ "Linux") {  -- First action --  x = NR+2  }  -- Second condition --  if (NR <= x) {  -- Second action --  print  }  }  Eg:  $ cat tst.awk  {  match($0,/^(\s\*(\S)\s\*)(.\*)/,a)  currHead = a[1]  currChar = a[2]  currTail = a[3]  }  currChar == "#" { indent = currHead }  currChar != "#" { indent = (prevChar == "#" ? currHead : indent) }  { printf "%s%s\n", indent, currTail; prevChar = currChar }  $ awk -f tst.awk file  # jkakjshkjh  \* drink (2 spaces \* 2 spaces)(non hash starting)  \* biscuit (1 space \* 2 spaces)(non hash starting)  \* paper (\* 1 space)(non has starting)  \* .. (many more lines) of non hash starting  \* tea (7 spaces \* 3 space)(non has starting)  # happy  \* cup (\* 1 space)(non has starting)  \* bat (2 spaces \* 2 spaces)(non hash starting)  \* scooter (1 space \* 2 spaces)(non hash starting)  \* .. (many more lines) of non hash starting  \* disk (7 spaces \* 3 space)(non has starting)  To help you understand the syntax, if I were writing the above in a C-like language then it'd be:  while ( read(FILENAME,line) ) { # awk does this for you  NR++; # awk does this for you  NF = split(line into $1, $2, $3, ... $NF); # awk does this for you  match(line,/^(\s\*(\S)\s\*)(.\*)/,a);  currHead = a[1];  currChar = a[2];  currTail = a[3];  if (currChar == "#") { indent = currHead; }  if (currChar != "#") { indent = (prevChar == "#" ? currHead : indent); }  printf "%s%s\n", indent, currTail; prevChar = currChar;  } # awk does this for you  and in fact you can duplicate that syntax in awks BEGIN section with:  BEGIN {  filename = ARGV[1]  ARGV[1] = ""  ARGC--  while ( (getline line < filename) > 0) ) {  nr++  nf = split(line,flds)  match(line,/^(\s\*(\S)\s\*)(.\*)/,a)  currHead = a[1]  currChar = a[2]  currTail = a[3]  if (currChar == "#") { indent = currHead }  if (currChar != "#") { indent = (prevChar == "#" ? currHead : indent) }  printf "%s%s\n", indent, currTail; prevChar = currChar  }  }  but see http://awk.freeshell.org/AllAboutGetline for why not to do that unless you have a very specific need. |
|  |

# awk '/pattern1/ || /pattern2/{print}'

|  |
| --- |
| awk '/pattern1/ || /pattern2/{print}'  Edit  To be fair, I like lhf's way better via /pattern1|pattern2/ since it requires less typing for the same outcome. However, I should point out that this template cannot be used for logical AND operations, for that you need to use my template which is /pattern1/ && /pattern2/ |

# The syntax of awk is:

|  |
| --- |
| awk 'pattern{action}' file  where the pattern indicates the pattern or the condition on which the action is to be executed for every line matching the pattern. In case of a pattern not being present, the action will be executed for every line of the file. In case of the action part not being present, the default action of printing the line will be done. Let us see some examples: |

# BEGIN block

|  |
| --- |
| BEGIN {awk-commands}  The begin block gets executed at program startup and is executes only once. This is good place to initialise variables. BEGIN is the AWK keyword and hence it must be in upper case. Please note that this block is optional. |

# Body Block

|  |
| --- |
| /pattern/ {awk-commands}  OR  Condition\_check/regexp{action}  The body block apply AWK commands on every input line. By default AWK execute commands on every line but we can restrict this by providing pattern. Note that there is no keyword for Body block |

# END BLOCK

|  |
| --- |
| END {awk-commands}  The end block gets executed at the end of program. END is the AWK keyword and hence it must be in upper case. Please note that this block is also optional.  Examples |

# EXAMPLES (BEGIN BLOCK)

|  |
| --- |
| Let us create a file marks.txt which contains the serial number, name of the student, subject name and number of marks obtained.  1) Amit Physics 80  2) Rahul Maths 90  3) Shyam Biology 87  4) Kedar English 85  5) Hari History 89  Now let us display the file contents with header by using AWK script.  [jerry]$ awk 'BEGIN{printf "Sr No\tName\tSub\tMarks\n"} {print}' marks.txt  When the above code is executed, it will produce the following result.  Sr No Name Sub Marks  1) Amit Physics 80  2) Rahul Maths 90  3) Shyam Biology 87  4) Kedar English 85  5) Hari History 89  At program startup AWK prints header from BEGIN block. Then in body block, it reads a line from a file and executes AWK's print command which just prints the contents on the standard output stream. This process repeats until file is exhausted. |

# EXAMPLE BODY BLOCK

|  |
| --- |
| Example  Consider we have a text file marks.txt to be processed and it has following content:  1) Amit Physics 80  2) Rahul Maths 90  3) Shyam Biology 87  4) Kedar English 85  5) Hari History 89  Let's display complete content of the file using AWK as follows:  [jerry]$ awk '{print}' marks.txt  On executing the above code, you get the following result:  1) Amit Physics 80  2) Rahul Maths 90  3) Shyam Biology 87  4) Kedar English 85  5) Hari History 89 |

# DUMPING VARIABLES

|  |
| --- |
| The --dump-variables[=file] option  It prints a sorted list of global variables and their final values to file. The default file is awkvars.out.  [jerry]$ awk --dump-variables ''  [jerry]$ cat awkvars.out  On executing the above code, you get the following result:  ARGC: 1  ARGIND: 0  ARGV: array, 1 elements  BINMODE: 0  CONVFMT: "%.6g"  ERRNO: ""  FIELDWIDTHS: ""  FILENAME: ""  FNR: 0  FPAT: "[^[:space:]]+"  FS: " "  IGNORECASE: 0  LINT: 0  NF: 0  NR: 0  OFMT: "%.6g"  OFS: " "  ORS: "\n"  RLENGTH: 0  RS: "\n"  RSTART: 0  RT: ""  SUBSEP: "\034"  TEXTDOMAIN: "messages" |

# FIELDS

If you are familiar with the Unix/Linux or do [bash shell programming](https://www.tecmint.com/category/bash-shell/), then you should know what internal field separator (IFS) variable is. The default IFS in Awk are tab and space.

$1 $2 $3 …..

## Example

|  |
| --- |
| # cat test.txt  Hare krishna hare krishna  $ awk '//{print $1 $2 $3 }' test.txt  Harekrishnahare  If you have noticed in the printed output, the field values are not separated and this is how print behaves by default.  $ awk '//{print $1, $2, $3; }' test.txt  Hare krishna hare |

One important thing to note and always remember is that the use of ($) in Awk is different from its use in shell scripting.

Under shell scripting ($) is used to access the value of variables while in Awk ($) it is used only when accessing the contents of a field but not for accessing the value of variables.

Awk also has a printf command that helps you to format your output is a nice way as you can see the above output is not clear enough.

## NF

NF is a predefined variable whose value is the number of fields in the current record.

|  |
| --- |
| #gauranga  This seems like a pretty nice example. |

No matter how many fields there are, the last field in a record can be represented by $NF. So, $NF is the same as $7, which is ‘example.’

## $0

The use of $0, which looks like a reference to the “zeroth” field, is a special case: it represents the whole input record. Use it when you are not interested in specific fields.

## EXAMPLE $1 and $0 and regex

|  |
| --- |
| $ awk '$1 ~ /li/ { print $0 }' mail-list  Amelia 555-5553 amelia.zodiacusque@gmail.com F  Julie 555-6699 julie.perscrutabor@skeeve.com F |

This example prints each record in the file mail-list whose first field contains the string ‘li’.

By contrast, the following example looks for ‘li’ in *the entire record* and prints the first and last fields for each matching input record:

|  |
| --- |
| $ awk '/li/ { print $1, $NF }' mail-list  Amelia F  Broderick R  Julie F  Samuel A |

## Nonconstant Field Numbers

A field number need not be a constant. Any expression in the awk language can be used after a ‘$’ to refer to a field. The value of the expression specifies the field number. If the value is a string, rather than a number, it is converted to a number. Consider this example:

|  |
| --- |
| #gauranga  awk '{ print $NR }' |

Recall that NR is the number of records read so far: one in the first record, two in the second, and so on. So this example prints the first field of the first record, the second field of the second record, and so on. For the twentieth record, field number 20 is printed; most likely, the record has fewer than 20 fields, so this prints a blank line. Here is another example of using expressions as field numbers:

|  |
| --- |
| file1  1  1 2  1 2 3  1 2 3 4  1 2 3 4 5  awk '{ print $NR }' file1 (NR is the number of records read so far)  1 (NR =1 so print $1)  2 (NR =2 so print $2 of 1 2)  3 (NR =3 so print $3 of 1 2 3)  4 (NR =4 so print $4 of 1 2 3 4)  5 (NR =5 so print $5 of 1 2 3 4 5) |

As mentioned in [Fields](https://www.gnu.org/software/gawk/manual/html_node/Fields.html#Fields), awk stores the current record’s number of fields in the built-in variable NF (also see [Built-in Variables](https://www.gnu.org/software/gawk/manual/html_node/Built_002din-Variables.html#Built_002din-Variables)). Thus, the expression $NF is not a special feature—it is the direct consequence of evaluating NF and using its value as a field number.

## Changing the Contents of a Field

The contents of a field, as seen by awk, can be changed within an awk program; this changes what awk perceives as the current input record. (The actual input is untouched; awk *never*modifies the input file.) Consider the following example and its output:

|  |
| --- |
| $ awk '{ nboxes = $3 ; $3 = $3 - 10  > print nboxes, $3 }' inventory-shipped  25 15  32 22  24 14  … |

The program first saves the original value of field three in the variable nboxes. The ‘-’ sign represents subtraction, so this program reassigns field three, $3, as the original value of field three minus ten: ‘$3 - 10’. (See [Arithmetic Ops](https://www.gnu.org/software/gawk/manual/html_node/Arithmetic-Ops.html#Arithmetic-Ops).) Then it prints the original and new values for field three. (Someone in the warehouse made a consistent mistake while inventorying the red boxes.)

When the value of a field is changed (as perceived by awk), the text of the input record is recalculated to contain the new field where the old one was. In other words, $0 changes to reflect the altered field. Thus, this program prints a copy of the input file, with 10 subtracted from the second field of each line:

|  |
| --- |
| $ awk '{ $2 = $2 - 10; print $0 }' inventory-shipped  Jan 3 25 15 115  Feb 5 32 24 226  Mar 5 24 34 228  … |

It is also possible to assign contents to fields that are out of range. For example:

|  |
| --- |
| $ awk '{ $6 = ($5 + $4 + $3 + $2)  > print $6 }' inventory-shipped  168  297  301  … |

We’ve just created $6, whose value is the sum of fields $2, $3, $4, and $5. The ‘+’ sign represents addition. For the file inventory-shipped, $6 represents the total number of parcels shipped for a particular month.

## Example print last but one and last fields

|  |
| --- |
| awk '{print $(NF - 1), $NF}' |

## Example muliple print vs single print

|  |
| --- |
| $ echo "a b c d\n1 2 3 4" | awk '{print $0; print NF}'  a b c d  4  1 2 3 4  4  $ echo "a b c d\n1 2 3 4" | awk '{print $0, NF}'  a b c d 4  1 2 3 4 4 |

# HOW TO reconstitute the record

|  |
| --- |
| $1 = $1 # force record to be reconstituted  print $0 # or whatever else with $0 |

It is important to remember that $0 is the *full* record, exactly as it was read from the input. This includes any leading or trailing whitespace, and the exact whitespace (or other characters) that separates the fields.

It is a common error to try to change the field separators in a record simply by setting FS and OFS, and then expecting a plain ‘print’ or ‘print $0’ to print the modified record.

But this does not work, because nothing was done to change the record itself. Instead, you must force the record to be rebuilt, typically with a statement such as ‘$1 = $1’, as described earlier.

Eg:

|  |
| --- |
| echo a b c d e f | awk '{ print "NF =", NF; NF = 3; print $0 }'  NF = 6  a b c |

**CAUTION:** Some versions of awk don’t rebuild $0 when NF is decremented

# NR

NR is the number of records read so far

# EXAMPLE OFS AND FS SET LATE, Setting of variables

|  |
| --- |
| >Humanl|chr16:86430087-86430726 | element 1 | positive  >Humanl|chr16:85620095-85621736 | element 2 | negative  >Humanl|chr16:80423343-80424652 | element 3 | negative  >Humanl|chr16:80372593-80373755 | element 4 | positive  >Humanl|chr16:79969907-79971297 | element 5 | negative  >Humanl|chr16:79949950-79951518 | element 6 | negative  >Humanl|chr16:79026563-79028162 | element 7 | negative  >Humanl|chr16:78933253-78934686 | element 9 | negative  >Humanl|chr16:78832182-78833595 | element 10 | negative |

TRIED

|  |
| --- |
| #gauranga  awk '{FS="|";OFS="\t"} {print $1,$2,$3,$4,$5}' |

OUTPUT

|  |
| --- |
| >Human|chr16:86430087-86430726 | element 1 |  >Human chr16:85620095-85621736 element 2 negative  >Human chr16:80423343-80424652 element 3 negative  >Human chr16:80372593-80373755 element 4 positive  >Human chr16:79969907-79971297 element 5 negative  >Human chr16:79949950-79951518 element 6 negative  >Human chr16:79026563-79028162 element 7 negative  >Human chr16:78933253-78934686 element 9 negative  >Human chr16:78832182-78833595 element 10 negative |

FAILED: Every line works fine except for the first line.

ANSWER:

|  |
| --- |
| FS and OFS are set too late to affect the first line, use something like this instead:  awk '{print $1,$2,$3,$4,$5}' FS='|' OFS='\t' ( append them after the script as VAR=VALUE. This is called post script variables)  You can also use this shorter version:  awk -v FS='|' -v OFS='\t' '$1=$1' (using -v)  OR  awk 'BEGIN{FS="|";OFS="\t"} {print $1,$2,$3,$4,$5}' (Using BEGIN)  Note that there is a significant difference between when -v and post-script variables are set. -v will set variables before the BEGIN clause whilst post-script setting of variables are set just after the BEGIN clause. |

# NR VS FNR

|  |
| --- |
| $ cat file1  a  b  c  $ cat file2  d  e  $ awk '{print FILENAME, NR, FNR, $0}' file1 file2  file1 1 1 a  file1 2 2 b  file1 3 3 c  file2 4 1 d  file2 5 2 e |

Keep in mind NR and FNR are awk built-in variables. NR - Gives the total number of records processed. (in this case both in a.txt and b.txt) FNR - Gives the total number of records for each input file (records in either a.txt or b.txt)

This means that the condition NR==FNR is only true for the first file, as FNR resets back to 1 for the first line of each file but NR keeps on increasing.

# EXAMPLE : Print number and File (NR vs count++)

|  |
| --- |
| $ for file in \*\*/\*; do file $file; done|awk 'BEGIN {count=1} !/directory/{print count++" "$0}'  1 file1.txt: empty  2 file2.txt: empty  3 file3.txt: empty  4 test: directory  5 file4.txt: empty  NOTE: if we use NR instead of count and the counting will not be in sequence  $ for file in \*\*/\*; do file $file; done|awk ‘!/directory/{print NR" "$0}'  1 file1.txt: empty  2 file2.txt: empty  3 file3.txt: empty  5 file4.txt: empty |

# Example: find any field beginning from 2 which contains word image

|  |
| --- |
| #gauranga  for file in \*\*/\*; do file $file; done|awk 'BEGIN {count=0} !/directory/{for (i=2;i<=NF;i++) {if ($i ~ /image/) {print count++,i,$i,$0}}}' |

# Example field

|  |
| --- |
| ls --full-time --time-style=+%Y-%m-%d-%H\_%M\_%S | awk '{print $6}'  2019-04-15-18\_14\_47  2019-04-15-18\_14\_47  2019-04-15-18\_14\_47  2019-04-15-18\_14\_47  2019-04-15-18\_16\_16  2019-04-09-01\_56\_41  2019-04-15-18\_15\_12 |

# EXAMPLE OF IF and matching each field with ~:

|  |
| --- |
| $ for file in \*\*/\*  do  f\_base=`echo $file | awk 'BEGIN {FS="."} {a=""; for (i=1;i<=NF;i++) {if (i==1) {a=$i;print i ":" a} else {a=a FS $i;print i ":" a}}; print a}'`;  echo $f\_base  Done  1:file1 (printed by if (i==1) {a=$i;print i ":" a})  2:file1.txt → printed by else {a=a FS $i;print i ":" a}  file1.txt printed by print a  1:file2  2:file2.txt  file2.txt  1:file3  2:file3.txt  file3.txt  1:file4  2:file4.txt  file4.txt  1:hare  2:hare.txt  hare.txt  1:Harinaam\_Monday\_ISKCON\_Chowpatty-qtFYjjo9LCc  2:Harinaam\_Monday\_ISKCON\_Chowpatty-qtFYjjo9LCc.mkv  Harinaam\_Monday\_ISKCON\_Chowpatty-qtFYjjo9LCc.mkv  1:test  test |

# RS

It represents (input)record separator and its default value is newline.

[jerry]$ awk 'BEGIN {print "RS = " RS}' | cat -vte

On executing the above code, you get the following result:

RS = $

$

# $0

It represents the entire input record.

|  |
| --- |
| [jerry]$ awk '{print $0}' marks.txt  On executing the above code, you get the following result:  1) Amit Physics 80  2) Rahul Maths 90  3) Shyam Biology 87  4) Kedar English 85  5) Hari History 89 |

# IGNORECASE

When this variable is set GAWK becomes case insensitive. Following simple example illustrates this:

|  |
| --- |
| [jerry]$ awk 'BEGIN{IGNORECASE=1} /amit/' marks.txt  On executing the above code, you get the following result:  1) Amit Physics 80 |

# EXAMPLE print

|  |
| --- |
| awk '$9 == 500 ' /var/log/httpd/access.log  awk '$9 == 500 {print} ' /var/log/httpd/access.log  awk '$9 == 500 {print $0} ' /var/log/httpd/access.log |

# REGEX NOT MATCHING

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

|  |
| --- |
| #guarangas  awk '!/regex/'  Example  Show all files which are not directories  for file in \*\*; do file $file; done|awk '!/directory/{print NR" "$0}'  OR (the below gives the right results)  for file in \*\*; do file $file; done|awk 'BEGIN {count=0} !/directory/{count++;print count" "$0}' |

# EXAMPLE find if contains words OR and AND

|  |
| --- |
| #gauranga  awk '/tom|jerry|vivek/' /etc/passwd  OR  awk '/tom/ || /jerry/ || /vivek/ {print}' /etc/passwd |

# Simply Arithmetic

|  |
| --- |
| You get the sum of all the numbers in a column:  awk '{total += $1} END {print total}' earnings.txt  Shell cannot calculate with floating point numbers, but awk can:  awk 'BEGIN {printf "%.3f\n", 2005.50 / 3}' |

# USE shell variable inside awk

|  |
| --- |
| $gauranga  search="android"; awk -v r="$search" |

# EXAMPLE using function inside awk

|  |
| --- |
| #gauranga  awk 'function green(s) { printf "\033[1;32m" s "\033[0m "}; /Recycle/{print green($0)}' IGNORECASE=1 \*\*/\*.java (working) |

# EXAMPLE substituting in awk gsub

|  |
| --- |
| $ search="91+[0-9]"; awk -v r="$search" '{if($0 ~ r && FNR==1){print "\n\033[1;31m"count++" : "FILENAME"\033[0m\n"; gsub(r,"\033[1;36m&\033[1;000m"); print FNR":"$0} else if(FNR==1){check=0} else if($0 ~ r) {if(check==0){print "\n\033[1;31m"count++" :"FILENAME"\033[0m\n";gsub(r,"\033[1;36m&\033[1;000m");print FNR":"$0;check=1} else {gsub(r,"\033[1;36m&\033[1;000m");print FILENAME":"FNR":"$0}}}' IGNORECASE=1 .vlist  0 :.vlist  2:137 91xxxxxxxxxx Mr. Vrajchandra Data  .vlist:3:342 91xxxxxxx Mr. Vraj Das  .vlist:4:201 91xxxxxxxxxx Mr. Hemant Thakur  .vlist:5:104 91xxxxxxxxx Mr. Bankim Rayba Data  .vlist:6:234 91xxxxxxxxxx Mr. Srikant Pool -GEV |

# EXAMPLE ADD new line after a line

|  |
| --- |
| $ awk '/android/;/android/{print "\n"}' IGNORECASE=1 .vlist  OR  $ awk '/android/{print;print "\n"}' IGNORECASE=1 .vlist  adb shell 'pm uninstall -k --user 0 com.sec.android.app.translator';  adb shell 'pm uninstall -k --user 0 com.sec.android.app.samsungapps';  adb shell 'pm uninstall -k --user 0 com.sec.android.app.kidshome'; |

# passing shell variables

|  |
| --- |
| How do I pass shell variables to awk command or script under UNIX like operating systems?  The -v option can be used to pass shell variables to awk command. Consider the following simple example,  root="/webroot"  echo | awk -v r=$root '{ print "shell root value - " r}'  In this example search file using awk as follows: |

# EXAMPLE How to adding all the numbers in 4th column and print the total value at the end.

|  |
| --- |
| awk 'BEGIN{s=0}{s=s+$3}END{print s}' db.txt  Output:  297 |

# How do I print all lines that have three (3) words only?

|  |
| --- |
| The awk command is well suitable for this kind of pattern processing text file. Awk set the variable called NF. It is set to the total number of fields in the input record. So if NF equal to three print the line. The syntax is as follows:  awk '{ if ( NF == 3 ) print } ' /path/to/input |

# EXAMPLE Gsub

|  |
| --- |
| Create a data file cat /tmp/data.txt  foo bar 12,300.50  foo bar 2,300.50  abc xyz 1,22,300.50  Type the following awk command:  awk '{ gsub(",","",$3); print $3 }' /tmp/data.txt  Sample outputs:  12300.50  2300.50  122300.50 |

# EXAMPLE Processing the delimited files using awk

|  |
| --- |
| You can also use awk command for same purpose:  $ awk -F':' '{ print $1 }' /etc/passwd  Output:  root  you  me  vivek  httpd  Where,  -F: - Use : as fs (delimiter) for the input field separator  print $1 - Print first field, if you want print second field use $2 and so on |

# \*\* EXAMPLE The default syntax to read a text file line-by-line using awk is as follows:

|  |
| --- |
| ### note '{ print }' '{ print $0 }' '1' all are same ###  awk '{ print }' /path/to/file  awk '{}1' /path/to/file  **awk '1' /path/to/file**  awk '{ print $0 }' /path/to/file  awk '{}1' /etc/passwd  awk '{print $0}' /etc/hosts  awk '{print}' /etc/hosts  awk '{ print $1 }' filename  To see first and fourth fields of the current record, enter:  awk '{ print $1, $3 }' filename  The commands "print" and "print $0" are identical in functionality. |

# \*\* EXAMPLE awk define and print variable

|  |
| --- |
| Create a variable called x and y:  awk 'BEGIN{x=3; y=4;}END{ print "x=" x " and y=" y}'</dev/null |

# \*\*EXAMPLE display line number

|  |
| --- |
| awk '{ print FNR " " $0 }' file  awk '{ print FNR " " $0 }' /etc/hosts  awk '{ print FNR "\t" $0 }' /etc/hosts |

# EXAMPLE print required lines

|  |
| --- |
| Print 42nd line using awk command  awk "NR==42{print;exit}" filename  OR print 13 to 42 lines:  awk "NR>=13{print} NR==42{exit}" /etc/passwd  Print 1 to 5 lines and number it on screen:  awk "NR>=1{print} NR==5{exit}" /etc/group | cat -n |

# Here are some ways to get variables in to awk:

|  |
| --- |
| You can use variable within the awk code, but its messy and hard to read:  variable="line one\nline two"  awk 'BEGIN {print "'"$variable"'"}'  line one  line two  variable="line one\nline two"; awk 'BEGIN {print "'"$variable"'"}'  variable="line one\nline two"; awk -v hare="$variable" 'BEGIN {print hare}'  This is the best way to do it is using -v option: (PS use space after -v or it will be less portable. Eks awk -v var= not awk -vvar)  awk -v var="$variable" 'BEGIN {print var}'  line one  line two  If you use variable in BEGIN block, it must be read with -v before BEGIN block.  To use variable elsewhere in awk you can read it after code like this:  echo "input data" | awk '{print var}' var="$variable"  or  awk '{print var}' var="$variable" file |

# \*\* spliting into files

|  |
| --- |
| #gauranga  awk '/XYZ/{a=$0}{if (a!="") print > a;}' sb\_full.txt |

# Conditional awk printing

|  |
| --- |
| The awk syntax is as follows:  awk 'condition { action }' filename  awk '1 == /match/ { action }' filename  awk '1 == /match/ { print $0 }' filename  awk 'pattern { action }' filename  If first field match to vivek print the entire line:  $ awk -F: '1 == /vivek/ { print $0}' /etc/passwd  OR,  $ awk -F: '/vivek/ { print $0}' /etc/passwd (if we use this it will search the match in the whole line |

# passing variable: Correct, awk won't recogize variables in / /. You can do:

|  |
| --- |
| Name="jony"  awk -v name="$Name" '$0 ~ name' file  Since print is awk's default behavior we can avoid using it here. |

# awk - Match a pattern in a file in Linux: ~ is the symbol used for pattern matching. The / / symbols are used to specify the pattern.

|  |
| --- |
| In one of our earlier articles on awk series, we had seen the basic usage of awk or gawk. In this, we will see mainly how to search for a pattern in a file in awk. Searching pattern in the entire line or in a specific column.  Let us consider a csv file with the following contents. The data in the csv file contains kind of expense report. Let us see how to use awk to filter data from the file.  $ cat file  Medicine,200  Grocery,500  Rent,900  Grocery,800  Medicine,600  1. To print only the records containing Rent:  $ awk '$0 ~ /Rent/{print}' file  Rent,900  ~ is the symbol used for pattern matching. The / / symbols are used to specify the pattern. The above line indicates: If the line($0) contains(~) the pattern Rent, print the line. 'print' statement by default prints the entire line. This is actually the simulation of grep command using awk.  2. awk, while doing pattern matching, by default does on the entire line, and hence $0 can be left off as shown below:  $ awk '/Rent/{print}' file  Rent,900  3. Since awk prints the line by default on a true condition, print statement can also be left off.  $ awk '/Rent/' file  Rent,900  In this example, whenever the line contains Rent, the condition becomes true and the line gets printed.  4. In the above examples, the pattern matching is done on the entire line, however, the pattern we are looking for is only on the first column. This might lead to incorrect results if the file contains the word Rent in other places. To match a pattern only in the first column($1),  $ awk -F, '$1 ~ /Rent/' file  Rent,900  The -F option in awk is used to specify the delimiter. It is needed here since we are going to work on the specific columns which can be retrieved only when the delimiter is known.  5. The above pattern match will also match if the first column contains "Rents". To match exactly for the word "Rent" in the first column:  $ awk -F, '$1=="Rent"' file  Rent,900  6. To print only the 2nd column for all "Medicine" records:  $ awk -F, '$1 == "Medicine"{print $2}' file  200  600  7. To match for patterns "Rent" or "Medicine" in the file:  $ awk '/Rent|Medicine/' file  Medicine,200  Rent,900  Medicine,600  8. Similarly, to match for this above pattern only in the first column:  $ awk -F, '$1 ~ /Rent|Medicine/' file  Medicine,200  Rent,900  Medicine,600  9. What if the the first column contains the word "Medicines". The above example will match it as well. In order to exactly match only for Rent or Medicine,  $ awk -F, '$1 ~ /^Rent$|^Medicine$/' file  Medicine,200  Rent,900  Medicine,600  The ^ symbol indicates beginning of the line, $ indicates the end of the line. ^Rent$ matches exactly for the word Rent in the first column, and the same is for the word Medicine as well.  10. To print the lines which does not contain the pattern Medicine:  $ awk '!/Medicine/' file  Grocery,500  Rent,900  Grocery,800  The ! is used to negate the pattern search.  11. To negate the pattern only on the first column alone:  $ awk -F, '$1 !~ /Medicine/' file  Grocery,500  Rent,900  Grocery,800  12. To print all records whose amount is greater than 500:  $ awk -F, '$2>500' file  Rent,900  Grocery,800  Medicine,600  13. To print the Medicine record only if it is the 1st record:  $ awk 'NR==1 && /Medicine/' file  Medicine,200  This is how the logical AND(&&) condition is used in awk. The records needed to be retrieved is only if it is the first record(NR==1) and the record is a medicine record.  14. To print all those Medicine records whose amount is greater than 500:  $ awk -F, '/Medicine/ && $2>500' file  Medicine,600  15. To print all the Medicine records and also those records whose amount is greater than 600:  $ awk -F, '/Medicine/ || $2>600' file  Medicine,200  Rent,900  Grocery,800  Medicine,600 |

# To get color output from awk, you can use this approach.

|  |
| --- |
| function red(s) {  printf "\033[1;31m" s "\033[0m "  }  function green(s) {  printf "\033[1;32m" s "\033[0m "  }  function blue(s) {  printf "\033[1;34m" s "\033[0m "  }  {  print red($1), green($2), blue($3)  } |

# AWK colors and ANSI graphics

ANSI code has always been pretty much of a minority interest and

(slightly old fashioned) black art, but it is useful & can be

very effective in conjunction with Awk. This FAQ just covers the

graphics parameters that allow coloured text, reverse video etc -

it does not cover the parameters which control the cursor movement

and on screen placement.

ANSI graphics parameters take the form

ESC[att;attm

where 'ESC' is the escape, 'att' represents one or more attributes

separated by ';' and 'm' terminates the code. In Linux (Unix) the

escape is \033, so:

echo -e '\033[5;41;1;37m \*\*\* STOP \*\*\* \033[0m'

will print the flashing message ' \*\*\* STOP \*\*\* ' in bright white

letters on a red background. The instruction '\033[0m' turns off all

attributes and returns the screen to its normal appearance.

# EXAMPLE OFS

|  |
| --- |
| To print the first and third columns, ie., the name and the expertise:  $ awk -F"," '{print $1, $3}' file1  Name Expertise  Deepak MQ Series  Neha Power Builder  Vijay CRM Expert  Guru Unix  10. The output shown above is not easily readable since the third column has more than one word. It would have been better had the fields being displayed are present with a delimiter. Say, lets use comma to separate the output. Also, lets discard the header record.  $ awk -F"," 'NR!=1{print $1,$3}' OFS="," file1  Deepak,MQ Series  Neha,Power Builder  Vijay,CRM Expert  Guru,Unix  OFS is another awk special variable. Just like how FS is used to separate the input fields, OFS (Output field separator) is used to separate the output fields. |

# How do we get the entire file content printed in other way?

|  |
| --- |
| $ awk '1' file1  Name Domain  Deepak Banking  Neha Telecom  Vijay Finance  Guru Migration  The above awk command has only the pattern or condition part, no action part. The '1' in the pattern indicates "true" which means true for every line. As said above, no action part denotes just to print which is the default when no action statement is given, and hence the entire file contents get printed. |

# How to omit the header record and get only the names printed?

|  |
| --- |
| $ awk 'NR!=1{print $1}' file1  Deepak  Neha  Vijay  Guru  The above awk command uses a special variable NR. NR denotes line number ranging from 1 to the actual line count. The conditon 'NR!=1' indicates not to execute the action part for the first line of the file, and hence the header record gets skipped. |

# awk - Match a pattern in a file in Linux

|  |
| --- |
| In one of our earlier articles on awk series, we had seen the basic usage of awk or gawk. In this, we will see mainly how to search for a pattern in a file in awk. Searching pattern in the entire line or in a specific column.  Let us consider a csv file with the following contents. The data in the csv file contains kind of expense report. Let us see how to use awk to filter data from the file.  $ cat file  Medicine,200  Grocery,500  Rent,900  Grocery,800  Medicine,600  1. To print only the records containing Rent:  $ awk '$0 ~ /Rent/{print}' file  Rent,900  ~ is the symbol used for pattern matching. The / / symbols are used to specify the pattern. The above line indicates: If the line($0) contains(~) the pattern Rent, print the line. 'print' statement by default prints the entire line. This is actually the simulation of grep command using awk.  2. awk, while doing pattern matching, by default does on the entire line, and hence $0 can be left off as shown below:  $ awk '/Rent/{print}' file  Rent,900  3. Since awk prints the line by default on a true condition, print statement can also be left off.  $ awk '/Rent/' file  Rent,900  In this example, whenever the line contains Rent, the condition becomes true and the line gets printed.  4. In the above examples, the pattern matching is done on the entire line, however, the pattern we are looking for is only on the first column. This might lead to incorrect results if the file contains the word Rent in other places. To match a pattern only in the first column($1),  $ awk -F, '$1 ~ /Rent/' file  Rent,900  The -F option in awk is used to specify the delimiter. It is needed here since we are going to work on the specific columns which can be retrieved only when the delimiter is known.  5. The above pattern match will also match if the first column contains "Rents". To match exactly for the word "Rent" in the first column:  $ awk -F, '$1=="Rent"' file  Rent,900  6. To print only the 2nd column for all "Medicine" records:  $ awk -F, '$1 == "Medicine"{print $2}' file  200  600  7. To match for patterns "Rent" or "Medicine" in the file:  $ awk '/Rent|Medicine/' file  Medicine,200  Rent,900  Medicine,600  8. Similarly, to match for this above pattern only in the first column:  $ awk -F, '$1 ~ /Rent|Medicine/' file  Medicine,200  Rent,900  Medicine,600  9. What if the the first column contains the word "Medicines". The above example will match it as well. In order to exactly match only for Rent or Medicine,  $ awk -F, '$1 ~ /^Rent$|^Medicine$/' file  Medicine,200  Rent,900  Medicine,600  The ^ symbol indicates beginning of the line, $ indicates the end of the line. ^Rent$ matches exactly for the word Rent in the first column, and the same is for the word Medicine as well.  10. To print the lines which does not contain the pattern Medicine:  $ awk '!/Medicine/' file  Grocery,500  Rent,900  Grocery,800  The ! is used to negate the pattern search.  11. To negate the pattern only on the first column alone:  $ awk -F, '$1 !~ /Medicine/' file  Grocery,500  Rent,900  Grocery,800  12. To print all records whose amount is greater than 500:  $ awk -F, '$2>500' file  Rent,900  Grocery,800  Medicine,600  13. To print the Medicine record only if it is the 1st record:  $ awk 'NR==1 && /Medicine/' file  Medicine,200  This is how the logical AND(&&) condition is used in awk. The records needed to be retrieved is only if it is the first record(NR==1) and the record is a medicine record.  14. To print all those Medicine records whose amount is greater than 500:  $ awk -F, '/Medicine/ && $2>500' file  Medicine,600  15. To print all the Medicine records and also those records whose amount is greater than 600:  $ awk -F, '/Medicine/ || $2>600' file  Medicine,200  Rent,900  Grocery,800  Medicine,600 |

# 3. awk has the getline command which reads the next line from the file.

|  |
| --- |
| $ awk '/Linux/{print;getline;print;}' file  Linux  Solaris  Once the line containing the pattern Linux is found, it is printed. getline command reads the next line into $0. Hence, the second print statement prints the next line in the file.  4. In this, the same thing is achieved using only one print statement.  $ awk '/Linux/{getline x;print $0 RS x;}' file  Linux  Solaris  getline x reads the next line into variable x. x is used in order to prevent the getline from overwriting the current line present in $0. The print statement prints the current line($0), record separator(RS) which is the newline, and the next line which is in x. - See more at: http://www.theunixschool.com/2012/05/different-ways-to-print-next-few-lines.html#sthash.1DZYmvl6.dpuf |

# o print only the line following the pattern without the line matching the pattern:

|  |
| --- |
| Same using awk:  $ awk '/Linux/{getline;print;}' file  Solaris  - See more at: http://www.theunixschool.com/2012/05/different-ways-to-print-next-few-lines.html#sthash.1DZYmvl6.dpuf |

# Print multiple lines are selecting

$ awk '$2=="KFC" {print; for(i=1; i<=4; i++) { getline; print}}' example.txt

Restaurant: KFC

City: NYC

State: NY

Address: 123 Madison Square

Phone: 911

The above command will get and print the consecutive 4 lines along with the current line because it was fed into a for loop.The search pattern $2=="KFC" will helps to get a particular line from the multiple lines.

# Gsub vs sub

|  |
| --- |
| Use gsub which does global substitution:  echo This++++this+++is+not++done | awk '{gsub(/\++/," ");}1'  sub function replaces only 1st match, to replace all matches use gsub.  OR  echo "This++++this+++is+not++done" | sed -re 's/(\+)+/ /g' |

# AWK: the substr command to select a substring

substr(s, a, b) : it returns b number of chars from string s, starting at position a. The parameter b is optional, in which case it means up to the end of the string.

Example:

|  |
| --- |
| every good  Let us save this data into a file called data.txt  Then, here are a few case examples:  Shell  awk '{print substr($1,1,1)}' data.txt #returns e  awk '{print substr($1,3) }' data.txt #returns ery  awk '{print substr($2,3) }' data.txt #returns od  awk '{print substr($0,7,2) }' data.txt #returns go |

Example 2:

|  |
| --- |
| Addr:192.168.1.135  awk '{print substr(column-number,start-point,end-point)}'  $ ifconfig eth0 | awk '/inet addr/{print substr($2,6)}'  192.168.1.135  $ ifconfig eth0 | awk '/inet addr/{print substr($2,6,3)}'  192 |

# Match function

|  |
| --- |
| $ awk 'BEGIN {  str = " \* In django we generall do migration after creating a django project and configuring the database in settings.py. Also ./manage.py runserver shows warning that migrations are pending. Basically there are many applications which are mentioned in INSTALLED APPS which need to setup their tables in database."  subs = "Two"  ret = match(str,/^(\s\*(\S)\s\*)(.\*)/,a)  print "a[1]::" a[1]  print "a[2]::"a[2]  print "a[3]::" a[3]  print "RLENGTH::"RLENGTH  print "RSTART::"RSTART  }'  a[1]:: \*  a[2]::\*  a[3]::In django we generall do migration after creating a django project and configuring the database in settings.py. Also ./manage.py runserver shows warning that migrations are pending. Basically there are many applications which are mentioned in INSTALLED APPS which need to setup their tables in database.  RLENGTH::312  RSTART::1 |

# EXAMPLE EXIT AFTER 1st RECORD

|  |
| --- |
| $ awk '/\\* #[^#]/{print}' README.md  \* # Django Documentation:  \* # 1.NEVER DO FIRST MIGRATION WITHOUT CREATING A CUSTOM “USER” MODEL and setting AUTH\\_USER\\_MODEL = custom\\_user.User in Settings.py  \* # 2. NEVER UPGRADE POSTGRESQL WITHOUT TAKING BACKUP.  \* # Github Documentation using MarkDown:  \* # FIRST COMMIT START  \* # FIRST COMMIT END  \* # SECOND COMMIT START  \* # SECOND COMMIT END  \* # THIRD COMMIT START  \* # THIRD COMMIT END  \* # FOURTH COMMIT THEORY START  \* # FOURTH COMMIT THEORY END  \* # FOURTH COMMIT START  \* # FOURTH COMMIT END  simha ~/DOC\_django\_basic  $ awk '/\\* #[^#]/{print;exit}' README.md  \* # Django Documentation: |

# AWK: Access captured group from line pattern

|  |
| --- |
| With gawk, you can use the match function to capture parenthesized groups.  gawk 'match($0, pattern, ary) {print ary[1]}'  example:  echo "abcdef" | gawk 'match($0, /b(.\*)e/, a) {print a[1]}'  outputs cd.  Note the specific use of gawk which implements the feature in question.  For a portable alternative you can achieve similar results with match() and substr.  example:  echo "abcdef" | awk 'match($0, /b[^e]\*/) {print substr($0, RSTART+1, RLENGTH-1)}'  outputs cd. |

# EXAMPLE awk print

|  |
| --- |
| NODE\_1  port 1  description blah  port 2  description blah blah  NODE\_2  port 1  description blah  port 2  description blah  NODE\_3  port 1  port 2  NODE\_4  port 1  port 2  NODE\_5  port 1  port 2  NODE\_6  port 1  description blahdy blah  port 2  description floop-a-doop |

|  |
| --- |
| awk '/^port/{port=$0}/^description/{print port;print $0}' nodes  port 1  description blah  port 2  description blah blah  port 1  description blah  port 2  description blah  port 1  description blahdy blah  port 2  description floop-a-doop |

|  |
| --- |
| awk '/^NODE/{print $0}/^port/{port=$0}/^description/{print port;print $0}' nodes  NODE\_1  port 1  description blah  port 2  description blah blah  NODE\_2  port 1  description blah  port 2  description blah  NODE\_3  NODE\_4  NODE\_5  NODE\_6  port 1  description blahdy blah  port 2  description floop-a-doop |

|  |
| --- |
| That description is actually wrong. What you want is: "If the line after port isn't description  don't print node and port either."  Code:  awk '/^NODE/{node=$0}/^port/{port=$0}/^description/{if(prev!=node){print node};print port;print $0;prev=node}' nodesNODE\_1  port 1  description blah  port 2  description blah blah  NODE\_2  port 1  description blah  port 2  description blah  NODE\_6  port 1  description blahdy blah  port 2  description floop-a-doop  The c-syntax  While read (filename, line){  NR++  If ($0 ~ '/^NODE/){  node=$0  }  If ($0 ~ /^port/){  port=$0  }  …  } |

# EAMPLE print array of match

|  |
| --- |
| Txt file:  NODE\_1  port 1  description blah  port 2  description blah blah  NODE\_2  port 1  description blah  port 2  description blah  NODE\_3  port 1  port 2  NODE\_4  port 1  port 2  NODE\_5  port 1  port 2  NODE\_6  port 1  description blahdy blah  port 2  description floop-a-doop  awk 'BEGIN{count=0}  match($0,/NODE/,a)  { <-- matters (use it above for not printing the matched line)  if(RSTART != 0){  print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"  for (i in a)  print i"-->"a[i]  count++;  print "count-->"count;  print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"  }  if (count >= 3)  {  exit  }  }' awksampledata5.txt  NODE\_1  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  0start-->1  0length-->4  0-->NODE  count-->1  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  NODE\_2  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  0start-->1  0length-->4  0-->NODE  count-->2  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  NODE\_3  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  0start-->1  0length-->4  0-->NODE  count-->3  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |

EXAMPLE awk replace part of text with previous lines USED in README.md

We have to use match when we want to use regex groups.

|  |
| --- |
| $ awk 'BEGIN{  count=0;  prevNR=0;  prevChar=""  }  match($0,/^(( +)(\\*)( +))([^#].\*)/,a){  if(prevNR!=0 && NR == prevNR+2){  print prevChar a[5]  prevNR = NR  }  else{  print;  prevChar = a[1];  prevNR = NR  }  }  !/^(( +)(\\*)( +))([^#].\*)/{  print;  }  /^ +\\* +#/{  prevChar = "";  }  ' testread.md  convert :  \* ## 2.Install the PostgreSQL Django adapter, psycopg2  \* <table><colgroup><col style="width: 100%" />4  \* <table><colgroup><col style="width: 100%" /></colgroup><tbody>1  \* <table><colgroup><col style="width: 100%" /></colgroup><tbody>2  \* <table><colgroup><col style="width: 100%" /></colgroup><tbody>3  to  \* ## 2.Install the PostgreSQL Django adapter, psycopg2  \* <table><colgroup><col style="width: 100%" />4  \* <table><colgroup><col style="width: 100%" /></colgroup><tbody>1  \* <table><colgroup><col style="width: 100%" /></colgroup><tbody>2  \* <table><colgroup><col style="width: 100%" /></colgroup><tbody>3 |

EXAMPLE RS (RECORD SEPERATOR)

|  |
| --- |
| $ gawk 'BEGIN {FS="\n" ; RS="here\n"} {printf "Row:%d Data:[%s %s %s]\n", NR, $1, $2, $3}' file  Row:1 Data:[1 2 3]  Row:2 Data:[a b c] |

# EXAMPLE AWK FS SLASH /

|  |
| --- |
| var="/home/vivek/foo/bar"  awk -F/ '{ print $4}' <<<"${var}"  OR  value=$(awk -F/ '{ print $4}' <<<"${PWD}")  echo "$value"  You can replace the variable name with the file name:  awk -F/ '{ print $4}' /path/to/file.txt  Where,  -F/ : sets the field separator, FS, to /. |

|  |
| --- |
| echo $1  echo $1 | awk -F/ '{print $(NF-1)}'  echo $1 | awk 'BEGIN{FS=OFS="/"}{NF--}1' |

# EXAMPLE match line between patterns using awk

|  |
| --- |
| Here is another approach  Include both patterns (default)  $ awk '/PAT1/,/PAT2/' file  PAT1  3 - first block  4  PAT2  PAT1  7 - second block  PAT2  PAT1  10 - third block  Mask both patterns  $ awk '/PAT1/,/PAT2/{if(/PAT2|PAT1/) next; print}' file  3 - first block  4  7 - second block  10 - third block  Mask start pattern  $ awk '/PAT1/,/PAT2/{if(/PAT1/) next; print}' file  3 - first block  4  PAT2  7 - second block  PAT2  10 - third block  Mask end pattern  $ awk '/PAT1/,/PAT2/{if(/PAT2/) next; print}' file  PAT1  3 - first block  4  PAT1  7 - second block  PAT1  10 - third block |

# Perl replace text bewteen two patterns and not matching the patterns

|  |
| --- |
| Here is another approach  Include both patterns (default)  $ awk '/PAT1/,/PAT2/' file  PAT1  3 - first block  4  PAT2  PAT1  7 - second block  PAT2  PAT1  10 - third block  Mask both patterns  $ awk '/PAT1/,/PAT2/{if(/PAT2|PAT1/) next; print}' file  3 - first block  4  7 - second block  10 - third block  Mask start pattern  $ awk '/PAT1/,/PAT2/{if(/PAT1/) next; print}' file  3 - first block  4  PAT2  7 - second block  PAT2  10 - third block  Mask end pattern  $ awk '/PAT1/,/PAT2/{if(/PAT2/) next; print}' file  PAT1  3 - first block  4  PAT1  7 - second block  PAT1  10 - third block |

|  |
| --- |
| Print lines between PAT1 and PAT2  $ awk '/PAT1/,/PAT2/' file  PAT1  3 - first block  4  PAT2  PAT1  7 - second block  PAT2  PAT1  10 - third block  Or, using variables:  awk '/PAT1/{flag=1} flag; /PAT2/{flag=0}' file  How does this work?  /PAT1/ matches lines having this text, as well as /PAT2/ does.  /PAT1/{flag=1} sets the flag when the text PAT1 is found in a line.  /PAT2/{flag=0} unsets the flag when the text PAT2 is found in a line.  flag is a pattern with the default action, which is to print $0: if flag is equal 1 the line is printed. This way, it will print all those lines occurring from the time PAT1 occurs and up to the next PAT2 is seen. This will also print the lines from the last match of PAT1 up to the end of the file.  Print lines between PAT1 and PAT2 - not including PAT1 and PAT2  $ awk '/PAT1/{flag=1; next} /PAT2/{flag=0} flag' file  3 - first block  4  7 - second block  10 - third block  This uses next to skip the line that contains PAT1 in order to avoid this being printed.  This call to next can be dropped by reshuffling the blocks: awk '/PAT2/{flag=0} flag; /PAT1/{flag=1}' file.  Print lines between PAT1 and PAT2 - including PAT1  $ awk '/PAT1/{flag=1} /PAT2/{flag=0} flag' file  PAT1  3 - first block  4  PAT1  7 - second block  PAT1  10 - third block  By placing flag at the very end, it triggers the action that was set on either PAT1 or PAT2: to print on PAT1, not to print on PAT2.  By placing flag at the very end, it triggers the action that was set on either PAT1 or PAT2: to print on PAT1, not to print on PAT2.  Print lines between PAT1 and PAT2 - including PAT2  $ awk 'flag; /PAT1/{flag=1} /PAT2/{flag=0}' file  3 - first block  4  PAT2  7 - second block  PAT2  10 - third block  By placing flag at the very beginning, it triggers the action that was set previously and hence print the closing pattern but not the starting one.  Print lines between PAT1 and PAT2 - excluding lines from the last PAT1 to the end of file if no other PAT2 occurs  This is based on a solution by Ed Morton.  awk 'flag{  if (/PAT2/)  {printf "%s", buf; flag=0; buf=""}  else  buf = buf $0 ORS  }  /PAT1/ {flag=1}' file  As a one-liner:  $ awk 'flag{ if (/PAT2/){printf "%s", buf; flag=0; buf=""} else buf = buf $0 ORS}; /PAT1/{flag=1}' file  3 - first block  4  7 - second block  # note the lack of third block, since no other PAT2 happens after it  This keeps all the selected lines in a buffer that gets populated from the moment PAT1 is found. Then, it keeps being filled with the following lines until PAT2 is found. In that point, it prints the stored content and empties the buffer. |

# WORD find one or more occurances

|  |
| --- |
| WORD  assA  [a-z] matches a or s or s  but  [a-z]{1,} matched one or more i.e ass  Microsoft Word doesn't have special symbol for greedy one or more, but {1,} is a good workaround.  ? represents single character  NOTE: ?{1,} will not match assA because it does not work in word. |

REGEX NEGATIVE LOOKAHEAD AND HOW TO MATCH BETWEEN TWO PATTERNS

|  |
| --- |
| Replace between two strings and closest (with repeting from wither end patt)  shortest match between two words is  QWWK.\*?KWWQ  But if we want to search between two patterns (like beginning or end of sentence etc) and at the same time dont want inbetween few words then use below  (?s)(^QWWK(?:(?!QWWK|KWWQ).)\*KWWQ\n) |

# REGEX ? vs \*

|  |
| --- |
| ? The question mark indicates zero or one occurrences of the preceding element. For example, colou?r matches both "color" and "colour".  \* The asterisk indicates zero or more occurrences of the preceding element. For example, ab\*c matches "ac", "abc", "abbc", "abbbc", and so on.  The big difference is that asterisk matches zero or more occurrences, while question mark matches zero or one occurrence. Compare these two examples:  $ printf "colour\ncolor\ncolouur\n" | egrep 'colou?r'  colour  color  $ printf "colour\ncolor\ncolouur\n" | egrep 'colou\*r'  colour  color  colouur  $ printf "TEXT \nTEXTS \nTEXTSS \nTEXS " | egrep 'TEXTS? '  TEXT  TEXTS |

# EXAMPLE print each letter as column

|  |
| --- |
| $ echo "radhanath swami videos" | awk 'BEGIN{FS=""}{for(i=1;i<=NF;i++)print $(i)}'  r  a  d  h  a  n  a  t  h    s  w  a  m  i    v  i  d  e  o  s |

# awk without printing newline

|  |
| --- |
| In awk, print automatically inserts a ORS after printing. ORS stands for "output record separator" and defaults to the new line. So whenever you say print "hi" awk prints "hi" + new line.  This can be changed in two different ways: using an empty ORS or using printf.  Using an empty ORS  awk -v ORS= '1' <<< "hello  man"  This returns "helloman", all together.  The problem here is that not all awks accept setting an empty ORS, so you probably have to set another record separator.  awk -v ORS="-" '{print ...}' file  For example:  awk -v ORS="-" '1' <<< "hello  man"  Returns "hello-man-".  Using printf (preferable)  While print attaches ORS after the record, printf does not. Thus, printf "hello" just prints "hello", nothing else.  $ awk 'BEGIN{print "hello"; print "bye"}'  hello  bye  $ awk 'BEGIN{printf "hello"; printf "bye"}'  Hellobye |

|  |
| --- |
| Your problem can be reduced to the following:  $ cat infile  11 12  21 22  $ awk '{ printf ($1 $2) }' infile  11122122  printf is for formatted printing. I'm not even sure if the behaviour of above usage is defined, but it's not how it's meant to be done. Consider:  $ awk '{ printf ("%d %d\n", $1, $2) }' infile  11 12  21 22  "%d %d\n" is an expression that describes how to format the output: "a decimal integer, a space, a decimal integer and a newline", followed by the numbers that go where the %d are. printf is very flexible, see the manual for what it can do. |

# OFS default , (comma) to space

$ awk '{ print $1, $2 }' infile

11 12

21 22

This prints the first and second field, separated by a space – and print does add a newline without us telling it to.

More precisely, "separated by the value of the output field separator OFS", which defaults to a space and is printed wherever we use , between two arguments. Forgetting the comma is a popular mistake that leads to no space between the record fields.

# BUild in Varibles: FS, OFS, RS, ORS, NR, NF, FILENAME, FNR

there are 8: FS, OFS, RS, ORS, NR, NF, FILENAME, FNR

# Basic awk template to begin

|  |
| --- |
| #!/bin/awk -f  ###################################  # BEGIN {} # Begin section  # {} # Loop section  # END{} # End section  ###################################  ## To help understand the awk syntax, if I were writing the above in a C-like language then it'd be:  ## while ( read(FILENAME,line) ) { # awk does this for you  ## NR++; # awk does this for you  ## NF = split(line into $1, $2, $3, ... $NF); # awk does this for you  ###################################  ## regarding new line, printf, ORS and OFS  ##  ## use printf "%s", var (if we want without new line) and print "%s\n", var with newline  ##  ## use print var (if we want by default new line or uses ORS output record separator)  ##  ## EG ORS= (blank) then print var will not add new line automatically :  ## awk -v ORS= '1' <<< "hello  ## man"  ## This returns "helloman", all together.  ##  ## EG ORS="-"(dash) then print var will not add new line automatically instead add - (dash) :  ## awk -v ORS="-" '1' <<< "hello  ## man"  ## Returns "hello-man-".  ##  ## print var1, var2, var3 (comma is replaced with OFS - output field separator)  ###################################  ## If Else on pattern matching  ##  ## classic way  ## awk '{if ($0 ~ /pattern/) {then\_actions} else {else\_actions}}' file  ## $0 represents the whole input record.  ##  ## based on the ternary operator syntax  ## selector ? if-true-exp : if-false-exp  ## Example:  ## awk '{print ($0 ~ /two/)?NR "yes":NR "No"}' <<<$'one two\nthree four\nfive six\nseven two'  ## 1yes  ## 2No  ## 3No  ## 4yes  ##  ## More ways  ## awk '/pattern/' filename  ## #prints all lines that contain the pattern. Since print is default action  ## awk '!/pattern/' filename  ## #prints all lines that do not contain the pattern.  ## # If you find if(condition){}else{} an overkill to use  ## awk '/pattern/{print "yes";next}{print "no"}' filename  ## # Same as if(pattern){print "yes"}else{print "no"}  ##  ## More ways  ## awk '/regexp/{print "true"; next} {print "false"}'  ## awk '{if (/regexp/) {print "true"} else {print "false"}}'  ## awk '{print (/regexp/ ? "true" : "false")}'  ##  ## more ways  ## /REGEX/ {action-if-matches...}  ## ! /REGEX/ {action-if-does-not-match}  ##  #######################################  ## print an array use for(i in array)  ##  ##for(i in a){  ## print "["i"]: "a[i]  ## }  ##  ## Because array in awk is like dictionaries in python. The index can be a string not just 0,1,2, etc  ## a['hare'] = "gauranga" (will add to the array a)  ## since indexes are not numeric so we have to use for(i in a) to retrieve all the elements  ###################################  ## length of an array (if gawk is installed) use length(a)  ##  ## awk 'BEGIN{a[1]=1; a[2]=2; print length(a); a[23]=45; print length(a)}'  ## 2  ## 3  ##  ## OR  ## k = 0  ## for(i in a){  ## print "["i"]: "a[i]  ## k++  ## }  ## print "length: "k  ##  #######################################  ## check for index in an array (index in array)  ##  ## wrong way  ## # Check if "foo" exists in a: Incorrect!  ## if (a["foo"] != "") …  ## A reference to an element that does not exist automatically creates that array element, with the null string as its value. So here it will create a["foo"] and lenght of a[] get's incremented  ##  ## right way  ## https://www.gnu.org/software/gawk/manual/gawk.html#Reference-to-Elements  ## To determine whether an element exists in an array at a certain index, use the following expression:  ## indx in array  ## This expression tests whether the particular index indx exists, without the side effect of creating that element if it is not present. The expression has the value one (true) if array[indx] exists and zero (false) if it does not exist.  ## Eg:  ## if (2 in frequencies)  ## print "Subscript 2 is present."  ####################################  ## match function how it works  ## $ awk 'BEGIN{  ## str="some text simple test";  ## match(str,/simple (test)/,a);  ## print "a[0]: "a[0];  ## print "a[1]: "a[1];  ## print "a[2]: "a[2];  ## print "length(a): "length(a)  ## k = 0  ## for(i in a){  ## print "["i"]: "a[i]  ## k++  ## }  ## print "length: "k  ## print "RLENGTH::"RLENGTH  ## print "RSTART::"RSTART  ## }'  ##  ## OUTPUT  ## a[0]: simple test  ## a[1]: test  ## a[2]:  ## length(a): 7  ## [0start]: 11  ## [0length]: 11  ## [1start]: 18  ## [1length]: 4  ## [0]: simple test  ## [1]: test  ## [2]:  ## length: 7  ## RLENGTH::11  ## RSTART::11  ###################################  ## match function only gets the first match. If we want to get all the matches in a line then use split  ## https://stackoverflow.com/questions/40569441/awk-match-multiple-matches  ## split(str, array, magic, magicarray )  ##  ## awk 'BEGIN{  ## str="Hare Krishna Hare Krishna Krsna Krsna Hare Hare"  ## split(tolower(str),myarray,/kr\w+/,separray)  ## for (i in separray){  ## print separray[i]  ## }  ## }'  ##  ## krishna  ## krishna  ## krsna  ## krsna  ##  ## using split and ignorecase (this is preffered as it shows the actual case)  ## awk 'BEGIN{  ## IGNORECASE=1  ## str="Hare Krishna Hare Krishna Krsna Krsna Hare Hare"  ## split(str,myarray,/kr\w+/,separray)  ## for (i in separray){  ## print separray[i]  ## }  ## }'  ## Krishna  ## Krishna  ## Krsna  ## Krsna  ##  ##  ## without using split (this is also preferred)  ## awk 'BEGIN{  ## str="Hare Krishna Hare Krishna Krsna Krsna Hare Hare"  ## while(match(tolower(str),/kr\w+/))  ## {  ## print substr(str,RSTART,RLENGTH);  ## str=substr(str,RSTART+RLENGTH)  ## }  ## }'  ###################################  ## split(str, array, magic, magicarray ) for magic to be regex use /whatever/ instead of quotes  ## For magic:  ##  ## when you use a non-empty string (quoted by "") "...", awk will check the length of the string, if it is single char, it will be used as literal string (they call it separator). However if it is longer than 1, it will be treated as a dynamic regex.  ##  ## when you use static regex, which means, in format /.../, no matter how long is the expression, it will be always treated as regex.  ##  ## That is:  ##  ## "." - literal "." (period)  ## "[" - literal "["  ## "{" - literal "{"  ## ".\*" - regex  ## /./ - regex  ## /whatever/ -regex  ##  ## Eg:  ## echo 192.168.0.1 | awk '{ split($0,a,/./); print a[1] }'  ## # nothing here, every char is a delimiter.  ##  ## whereas  ## awk 'BEGIN{  ## str="192.168.0.1"  ## split(str,myarray,/\./)  ## for (i in myarray){  ## print myarray[i]  ## }  ## }'  ## 192  ## 168  ## 0  ## 1  ##  ##  ######################################  ## Case in-sensitive search using tolower() or IGNORECASE=1 (!=0 means to ignore the case on all regular expressions)  ##  ## Case sensitive match  ## awk 'BEGIN{  ## str = "Hare Krishna Hare Krishna, Krishna Krishna Hare Hare"  ## ret = match(str,/(krishna)/,a)  ## print "ret: "ret  ## for(i in a) print "["i"]: "a[i]  ## print "RLENGTH::"RLENGTH  ## print "RSTART::"RSTART  ## }'  ## ret: 0  ## RLENGTH::-1  ## RSTART::0  ## Case insesitive match using tolower()  ## $ awk 'BEGIN{  ## str = "Hare Krishna Hare Krishna, Krishna Krishna Hare Hare"  ## ret = match(tolower(str),/(krishna)/,a)  ## print "ret: "ret  ## for(i in a) print "["i"]: "a[i]  ## print "RLENGTH::"RLENGTH  ## print "RSTART::"RSTART  ## }'  ## ret: 6  ## [0start]: 6  ## [0length]: 7  ## [1start]: 6  ## [1length]: 7  ## [0]: krishna  ## [1]: krishna  ## RLENGTH::7  ## RSTART::6  ##  ## Case insesitive match using IGNORECASE  ## awk 'BEGIN{  ## IGNORECASE=1  ## str = "Hare Krishna Hare Krishna, Krishna Krishna Hare Hare"  ## ret = match(str,/(krishna)/,a)  ## print "ret: "ret  ## for(i in a) print "["i"]: "a[i]  ## print "RLENGTH::"RLENGTH  ## print "RSTART::"RSTART  ## }'  ## ret: 6  ## [0start]: 6  ## [0length]: 7  ## [1start]: 6  ## [1length]: 7  ## [0]: Krishna  ## [1]: Krishna  ## RLENGTH::7  ## RSTART::6  BEGIN{  FS=": "  }  {  s=NR " :: "$0" ## "  g=""  for(i=1;i<NF+1;i++)  {  if(i == NF){  g=g $i  }  else{  g=g $i" @@ "  }    }  print s g  }  END{  }  #Input:  #BookmarkBegin  #BookmarkTitle: Part II. Major Components  #BookmarkLevel: 1  #BookmarkPageNumber: 272  #output:  #NR :: $0 ## $i @@ $NF  #1 :: BookmarkBegin ## BookmarkBegin  #2 :: BookmarkTitle: Part II. Major Components ## BookmarkTitle @@ Part II. Major Components  #3 :: BookmarkLevel: 1 ## BookmarkLevel @@ 1  #4 :: BookmarkPageNumber: 272 ## BookmarkPageNumber @@ 272 |