Table of Contents

awk
standard Unix utility which has its own script language to provide data extraction, transformations, and reports. The GNU version of awk is gawk.
awk is based on C/very C-like syntax. It's very much different, but the syntax is borrowed in a lot of ways. Like sed, it shares a lot of similarities in that it operates on text-based data and is used for manipulating said data. awk is better for doing things like reports, summaries, extracting specific bits of data. sed is better at cosmetically transforming text (redacting, deleting, etc). Still, awk is more powerful!

- Text files are treated as lines of data records, each having many fields.
- Whitespace (spaces and/or tabs) is the default delimiter between fields. The delimiter can be changed by setting FS, which is the field separator.

Things to know about awk:

1. By default, awk has the same main input loop logic as sed. sed reads a line of input from the input stream, and runs the entire script against that one line. Then, when the script bottoms out, it prints the script of the pattern space. Then it starts over again. awk does the same thing (for the most part), but divides the data into records and fields. Records are lines, and lines are records. The record separator by default is just a new line character (most text based data is line oriented).

2. Once it takes that record, it uses a field separator which is defined by FS. The corresponding separator is called RS. FS by default is any amount of white space. That's what it's using to tokenize the records into fields. These don't have to be character literals.

```
FS=","
FS="[:-]"
field 1: field 2-field 3
FS="[ \t]+" -> default
```

3. awk/gawk is the same thing. gawk is the GNU (Linux version) of awk. There's a million different versions of awk, it doesn't matter which one you use on a linux machine. This is because it's just like vi/vim; Merely a symbolic link to the other.

The actions to be performed can be in either the command line directly, or can be a program script inside a programFile.

Syntax:

```
      awk options' program' file1 . .  .
      awk options -f programFile file1 . . .
      awk options -f programFile -
```

The third option tells awk that the input comes from stdin.
There can be 0 or more input files that it operates on; if you give it 0, then it reads it from stdin.
The hyphen means: standard input -> "I want to read from stdin"
This isn't a bash/linux feature, just one certain utilities support.

An awk program is a series of pattern action pairs:

```
      condition1 {action1}
      condition2 {action2}
      . . .
```

If the condition is true, the corresponding action is executed.
If the condition is omitted, the action is unconditionally executed.
Each action can be a series of commands.
The action pairs above are optional!

```
      {action1}
```

This is an unconditional option block, it will operate no matter what with no condition specified. You can have a condition with no action too!

Example:

```
awk 'NR == 4' inventory.txt #If we want to print the 4th line
```

we did {. . .} (no action) Means: print $0 (entire record)
The default action is to print the whole record.

```
awk '1' inventory.txt
```

is a condition that evaluates to true in a boolean context. All records will hit this. It'll be true and print the entire thing.

Condition with no action:

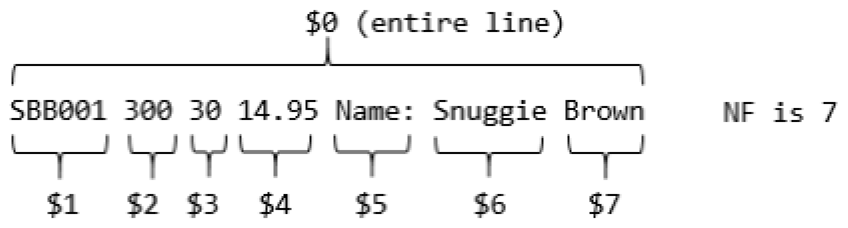
```
      awk '/error/' file.txt #prints lines containing error
```

No condition, only action:

```
      awk '{ print $1 }' file.txt #prints first field of every line
```

Neither:

```
      awk '{}' file.txt #does nothing for each line
```



```
$ awk '{ print $1 $4 }' inventory.txt
PPF0019.95
SBB00114.95
SBG00214.95
BOM00129.95
MCW00112.45
. . .
$ awk '{ print $1, $4 }' inventory.txt
PPF00 19.95
SBB00 114.95
SBG00 214.95
BOM00 129.95
MCW00 112.45
. . .
#The comma is taking the value of the variable OFS and inserts it
#like: { print $1 OFS $4 } OFS -> defaults to space
#Writing that instead of $1, $4 would create the same output

$ awk '{ printf("%s %s\n", $1, $4) }' inventory.txt
PPF001 9.95
SBB001 14.95
SBG002 14.95
BOM001 29.95
. . . #the same as above

$ awk -v OFS=x '{ print $1 OFS $4 }' inventory.txt
PPF001x9.95
SBB001x14.95
SBG002x14.95
BOM001x29.95
. . .
#Sets OFS to 'x'

$ awk -v ORS=x '{ print $1, $4 }' inventory.txt
PPF001 9.95xSBB001 14.95xSBG002 14.95xBOM001 . . .
#Sets ORS to x
```

```
awk$ head -n 3 inventory #by default a record is a line
PPF001 100 10  9.95 Name: Popeil Pocket Fisherman
SBB001 300 30 14.95 Name: Snuggie Brown
SBB002 300 30 14.95 Name: Snuggie Green

$ gawk '{print $1, $4}' inventory.txt #print field 1 & 4
PPF001 9.95
SBB001 14.95
SBB002 14.95
```

Some functions require you to use parenthesis:

```
$ gawk '{print($1, $4)}' inventory.txt
```

For the print command only: The comma isn't just used as a list separator, it has a special meaning. It expands to the output field separator, which by default is a space. So the comma injects a space. In awk string concatenations are performed simply by having strings adjacent to eachother (the space character is the string concatenation).

```
$ awk '{ print "sam" "silvestro" }' inventory.txt
samsilvestro
. . .
```

The most common conditions are pattern matches- similar to sed patterns; however, awk also makes it easy to apply a pattern to a particular field.

```
     $k ~ pattern
     $k !~ pattern
```

As far as doing pattern matches, awk uses '~' for the pattern binding operator.
If you had: name ~ /^[Ss]/ #does the value in name begin with s

There are no multi-line comments in awk.

NF (number of fields) and NR (number of records) get updated every time. Just because you have data all in the same format, doesn't mean it'll have the same number of fields every time. So if you want to know the number of fields, look at NF.
NR is basically a line counter when the default record separator is a newline, and records are lines, then you're literally doing a line count.
-> $0 represents the entire record

☆ awk is probably not the best for performance focused results. It's not meant to be fast but it's still useful.

☆ awk is in extended regex mode because that's all it can do- you can't turn it to basic regex mode or anything else.

☆ There is no switch to activate like with sed/grep (sed -E, or grep-E)

```
awk$ ~/genwords.py 123 128
#output prints the literal words for numbers
~/genwords.py 500 | awk 'NR == 100 { print $0 }'
one hundred
~/genwords.py 500 | awk ‘NR == 123 { print $0 }’
one hundred twenty-three #3 fields this tokenized on whitespace
~/genwords.py 500 | awk ‘NR == 123 { print $1 }’
one
~/genwords.py 500 | awk ‘NR == 123 { print $2 }’
hundred
~/genwords.py 500 | awk ‘NR == 123 { print $3 }’
twenty-three

#The issue is we don't know how many fields there are in the product
#description; This is where NF (number of fields) comes in handy.
$ cat >example5
⌜
  {
    printf("%6s ", $1); #output the product ID
    for (i=6; i <= NF; i++) #output each word in the product description.
     {
       printf("%s ", $i); #print a word from the description
     }
    printf("\n");
  }

                                                                        ⌟
```

Every single record will fire against this block
%6s -> 6 character wide string that represents the character ID
i=6 -> starts until i=<NF inclusive
which is counting the number of fields in, for example:
PPF001 Popeil Pocket Fisherman
PPF001 = field 1
Popeil = field 2
and so on

Again awk (creators of C) has a lot of C overlap. In awk, you don't need to initialize variables since variables don't have any data types innately, it's based on the context

Range Patterns

awk supports ranges of lines as a pattern, just like sed.

```
/pat1/,/pat2/
```

Given: /printf(/,/)/ {p; d}
sed will not check the same pattern space to see if it satisfies both the first and second part. It'll activate '/printf(/,/);', but won't check if {p; d} deactivates it.

If we also had
```
x=3;
a=x+3;
malloc(sizeof(int));
```

we would've printed /printf(/,/)/ {p; d} still. awk doesn't suffer from the same issue.
Addresses in sed are more concrete and rigid: either line numbers or patterns which indicate a certain substring is within the patternspace. There are no logical conditions or computations.

```
sed -n '3p'            #print 3rd line
sed -n '/cat/p'        #print lines with "cat"
sed -n '3,10p'         #print lines 3 to 10
sed -n '/cat/,/dog/p' #print from first "cat" to first "dog"
sed -n '3,/cat/p'      #print from 3rd line to 1st line with "cat"
```

In awk, it's more abstract and flexible than just evaluating "true" or "false". awk can handle logical operators, expressions, and the range conditions are more powerful.

```
NR == 3, NR == 10 { ... } #while num of records are between
                          #3 and 10 do whatever

/pat1/,/pat2/ {...} #shorthand for "look at the whole record
                    #and see if it contains thing"
```

- -	sed	awk
line numbers	yes	yes
pattern matching	yes	yes
field matching	no	yes
logical conditions	no	yes
arithmetic	no	yes
dynamic	no	yes

```
awk '//,//
```

In sed, the same pattern space can't activate the starting and ending address awk '/start/,/end'
In awk, that's not the case. In sed it'd stay active until some separate line contained the word 'end'.
Whenever you have just the forward slashes as an address, it's taking the whole record and checking to see if this pattern matches the entire thing.
Another difference is that sed, by default, prints everything out. Once the script bottoms out, it just prints the pattern space to the screen unless you use 'sed -n' to suppress default output.
awk will only print what you explicitly tell it to print.

```
$ awk '/Snuggie/ {print $1, $4}' inventory.txt
SBB001 14.95
SBG002 14.95

$ awk '$0 ~ /Snuggie/ {print $1, $4}' inventory.txt
SBB001 14.95
SBG002 14.95

$ awk '/Snuggie/ && /Brown/ {print $1, $4}' inventory.txt
SBB001 14.95
#Doing '$0 ~ ' results in the same output..

#What if we wanted to print the data associated with the records were
#the unit price ends in $0.95?
$ awk '$4 ~ /\.95$/ {print $1, $4}' inventory.txt
PPF001 9.95
SBB001 14.95
SBG002 14.95
BOM001 29.95
```

```
$ awk '$4 ~ /\.95$/ {print $1, $4}' inventory.txt
MCW001 12.45
#Displays fields that don't end with $0.95
#Changing it to {print $0} OR {print} simply prints the entire thing
#Output: MCW001 70 10 12.45 Name: Miracle Car Wax
```

Using a Program File

The -f switch is used to specify a program file which allows for more complex capabilities. awk provides C-like if, counting for, while, and printf action commands. It also supports a for in to iterate over the contents of an array.
Arrays in awk are associative arrays, not traditional; hashmaps and dictionaries are closer to awk arrays.

Some special conditions:

BEGIN ->	Executes the action before the records are read. In the actions, we typically initialize variables and print column headings.
END ->	Executes the action after the records are read. It's very common to print totals in the action corresponding to an END condition.

The awk arithmetic operators are from the C programming language. The type of comparison (numeric or string) is based on the operands. If both are numeric, a numeric comparison is done. Otherwise a string comparison is used.

People tend to overinitialize variables in awk!
In the END block, it's common to; print averages, totals, other calculations, other summary stuff.
BEGIN prints column headers.

Examples:

```
BEGIN
{
  printf("%-6s %4s %-10s\n", "ID", "QTY", "UNIT PRICE")
}
$2 > 200 { printf("%6s %4d %8.2f\n", $1, $2, $4) }
```

```
$ awk -f example4.awk inventory.txt
ID    QTY UNIT PRICE
SBB001 300 14.95
SBG002 400 14.95
NHC001 300 9.95

$ awk 'BEGIN { print "hello, world" }' inventory.txt
hello, world

$ awk 'BEGIN { print "Hello, world" }' asfd fdsgss
hello, world
#Still fires because the begin block fires super, super early
```

One common mistake:

In the begin block, when you try to reference specific fields such as:

```
$ awk 'BEGIN { print "field one=", $1 }' inventory.txt
field one= #Prints nothing because in the begin block we haven't
           #we haven't read anything so we don't know if it exists yet
```

Associative Arrays

awk supports associative arrays (i.e hash tables). The key for an associative array can be a character string. To assign a value:

```
      array[key] = value;
```

To check whether an entry exists:

```
      if (key in array)
            doSomething;
```

To iterate over the keys of the array:

```
      for (key in array)
            doSomething;
```

☆ The semicolons in awk aren't necessary. The only time you need them is if you're combining more than one statement (just like in bash)

Associative arrays/hashmaps/dictionaries are all the same syntactically. It looks identical to "regular" arrays:

```
      array[key] = value;
      operator[key to put in operator]
```

The backend of the data structure is implemented much differently.

A regular array is represented as a continuous block of memory where each element is the same size and data type (16 integers consecutively stored,..)
In this case, even though the syntax in which we access the values is identical, on the backend it's very different. The key that we give it (doesn't matter what key) like array[3] are always converted to strings. It'll take that number and convert it to an equivalent string (array["3"], array["name"], etc).

array[3.14352] -> floats are converted to their equivalent string
array["3"] -> array[3]

So what happens is (transcribed from lectures - in my court reporter era!)
It uses a hash function to take the key. You've got a string, it takes the hash of the string, and then that hash is mapped to the bucket. You have a variety of buckets, and in each of those buckets, stores its values in. Each bucket will be represented because it's essentially a linklist. It'll take the hash of element 3 to tell it what value to store it to and append it to the end of the list.

awk arrays are more flexible than traditional arrays in that they can mix data types, have no growth issues, and don't require continuous memory. One downside is that you don't know anything about the order or mapping of said array. For this, awk has a 'for (key in array) do Something; loop' The values that come out of this aren't in any recognizable order, and it seems random.

There's an "in" keyword in awk with 2 separate meanings: If you use it as part of an expression, it's the membership operator. "Hey is this key defined in the array/does this exist in the array/if so- do something!"

```
BEGIN
{
  printf("%-6s %4s %-10s %-12s\n", "ID", "QTY", "UNIT PRICE", "GROSS PRICE");
  total = 0;
}
{
  if ( $2 > 100 && $4 > 10.0 )
   {
     gross = $2 * $4;
     printf("%6s %4d %8.2f %10.2f\n", $1, $2, $4, gross);
     total = total + gross;
   }
}
END { printf("%s-6s %4s %-10s %10.2f\n", " ", " ", " ", total); }
```

┘

Output:

```
ID      QTY  UNIT PRICE  GROSS PRICE #Header came from the begin block
SBB001  300    14.95       4485.00    #These lines come from the loop
SBG002  400    14.95       5980.00
SSX001  150    14.95       4492.50
                          14957.50
```

┌

```
  BEGIN
  {
   printf("%-6s %4s %-10s %-12s\n", "ID", "QTY", "UNIT PRICE", "GROSS PRICE");
   #total = 0; -> by default 0 if used in an int context
   #so we can comment it out and get the same result
   #the hint is in the last print statement -> $10.2f is a float point
  }
  { $2 > 100 && $4 > 10.0
    {
     gross = $2 * $4;
     printf("%6s %4d %8.2f %10.2f\n", $1, $2, $4, gross);
     total += gross;
    }
  END { printf("%s-6s %4s %-10s %10.2f\n", " ", " ", " ", total); }
```

┘

If this was rewritten like 'print("   ", total)', assuming we never hit the math block, it would print an empty string. When we do it's still being used as an int due to 'total += gross'.

```
$ getent passwd | less
lightdm:x:108:115:Light Display Manager:/var/etc. . .
dnsmasq:x:109:65534:dnsmasq,,,:/var/. . .
#field 1 = username         field 2 = shadowpassword (will be x)
#field 3 = user id          field 4 = group id
#field 5 = gcode field      field 6 = home directory of user
#field 7 = login

$ getent passwd | less
$ getent passwd | grep ssilvestro
72:ssilvestro:x:7843:1000:ssilvestro:/home/etc . . .

$ cat >exer1.awk
BEGIN { FS = ":" }
$4 == 1000 {print $1;}
CTRL-D
```

Alternatives:

```
$ getent passwd | awk '$4 == 1000 {print $1}' FS=":" -

$ getent passwd | awk 'BEGIN{FS=":"}$4==1000' #shortest

$ getent passwd | awk -v FS=":" '$4==1000' #diff way to specify FS

$ getent passwd | awk 'BEGIN{FS=":"}$4==1000 { print $0 }' #longer

#not utilizing awk correctly:
$ getent passwd | awk 'BEGIN{FS=":"}{if($4==1000){print $0}}'
```

By default, records are lines.
By default, the field separator is any amount of white space.

```
getent passwd | awk -F: '$4 == 1000 { print $1 }' | sort
#Says: If field 4 equals 1000, print field 1 (names)

getent passwd | awk -F: '$4 == 1000 { print $1 }' FS=":" | sort
#You can pass variables through awk as if they're filenames
#awk can tell you're specifying a variable value and will recognize
#that's a variable assignment and change the separator to ':'
```

Example:

```
$ cat > example7

 ⌈
   BEGIN { blankCount = 0; commentCount = 0 }
    /^\/\*/,/\*\// { commentCount++ }
    /^[ \t]*\/\// { commentCount++ }
    /^[ \t]*$/ { blankCount++ }
   END
   {
    print "Total Lines:", NR
    print "Comment lines:", commentCount;
    print "Blank lines:", blankCount;
    print "Code:", NR - commentCount - blankCount;
   }
                                                  ⌋

$ awk -f example7 cs1713p0.c
Total Lines: 77
Comment lines: 26
Blank lines: 9
Code: 42
```

#When you hit the end-block the values of any variables you were using
#are available to you. They don't disappear/reset.

-> BEGIN { blankCount = 0; commentCount = 0 }
Can also do { blankCount = commentCount = 0 }

-> /^\/*/,/*\//
Does the line begin with a forward slash or an asterisk? End when we encounter one or the other.

-> /^[\t]*$/
What if it begins with a space or comment line?
Empty/blank lines will count too.

The other use is:

If you want to remove something in an array in awk, you can't just do myArr[3] = ""/0; or something.
You have to delete myArr[3], because an empty string or '0' is still considered defined.

So:

To check whether an entry exists -> if (key in array) doSomething;
To iterate over the keys of the array -> for (key in array) doSomething;

```
 $1 == "ORDER" && $2 == "ITEM"
{
  if ($3 in invM)
   invM[$3] += $4;
  else
   invM[$3] = $4;
}END{
 for (key in invM)
 print key, invM[key]; }
```

#Since we're only interested in items beginning with "order item"
#grep -color "^ORDER ITEM " invCommand.txt

Doing this a different way:

```
 ⌈
  $1 == "ORDER" && $2 == "ITEM"
  {
    if($3 in invM)
    {
     invM[$3] += 44
    }
    else
    {
     invM[$3] = $4
    }
  }END{
    for(key in invM)
     {
      print key, invM[key]
     }
  }
                                    ⌋
```

```
$ awk -f example8.awk invCommand.txt
XXX001 20
SBG002 410
SVC001 3
APC001 1
. . .
```

Depending on what version of awk you're using- it could implement the hashing function very differently.

```
$ awk -f example8.awk invCommand.txt | shuf
#shuf randomizes lines, data is correct -> order of lines is different.
```

How can we make the above program smaller?

$1 == “ORDER” && $2 == “ITEM” -> /^ORDER ITEM /

So:

```
/^ORDER ITEM /
{
 invM[$3] =+ $4 #gives numeric context
} END etc . . .
```

Example 9: Shell Script
In this example, the output from awk will be a shell script.
It's very common to have core dump files named "core" taking up a lot of space throughout your directories. As a system admin, you would use "locate core" to find the files which would give us a huge result containing many files. To simplify, we'll stimulate the use of locate by using find.
First find core files:

```
$ find ~ -name "*core*"
$ find ~ -name "*core*" -type f
/home/ssilvestro/courses/core
/home/ssilvestro/vim/share/vim/vim91
config_core.vim
```

core file size (blocks, -c) 0
The fox machines are set so 0 bytes are allocated to core dump files (as in; they're not created).

When one of your processes exits abnormally on most systems by default, a core dump file is created.
Lets say you get a segfault error. What will happen is a file named core in the current directory will be created, and what it contains really depends. You can put limits on the size- at minimum they'll have the execution context, the instruction pointer, the faulting address, faulting instruction, etc (register stuff). You'll probably have a stack backtrace as well. If you set it to unlimited, it'll dump the entire thing to the core space and that would be disgustingly fucking large and destroy your storage space (lmao).
You can attach to coredump files

```
gdb -c core
```

At the time segfault occurred - you can look at the memory/processes/etc where the issue occurred.

Example:

```
BEGIN { count = 0 }
/^\/home\/ssilvestro\// && /\/core$/
{
 print "rm", $0
 count++
} END { print "echo removed", count, "files"}
```

```
$ find ~ -name "*core*" -type f | awk -f example9.awk
rm /home/ssilvestro/courses/core
rm /home/ssilvestro/working/core
rm /home/ssilvestro/core
echo removed 3 files
#We put asterisks because just "core" would include non-core files
```

Passing Arguments Into awk Code

We can pass variable values into awk by specifying:

```
     awk options 'program' -v 'var=value' file1
```

The awk function match returns true if the functions in the first argument matches the pattern specified in the second argument.

In awk there's 4 ways to pass an argument:

```
getent passwd | tail #alias tail='tail -n 5'

getent passwd | awk '$4 == 1000 { print $ 1}'
getent passwd | awk -F: '$4 == 1000 { print $ 1}'

getent passwd | awk 'BEGIN { FS=":"}; $4 == 1000 { print $ 1}'
#modifies the script

getent passwd | awk '$4 == 1000 { print $ 1}' FS=":"

getent passwd | awk -v FS=":" '$4 == 1000 { print $ 1}' FS=":"
#only for gawk

getent passwd | awk '$4 == 1000 { print $ 1}' FS=":"
getent passwd | awk -v FS=":" '$4 == 1000 { print $ 1}' FS=":"
```

What's the difference between these two?

The first one is the oldschool way, but the begin block fires before any input validation is performed, and doesn't check if they exist/they're readable. That's what -v is for; it makes the variable assignment available from the very start so the begin block has access to it.

```
$ awk 'BEGIN { print "name =", name}'
name =
$ awk 'BEGIN { print "name =", name=Sam}'
name = #still blank bc begin fires before anything else
$ awk -v name=Sam 'BEGIN { print "name =", name}'
name = Sam
```

```
0. -F: -> FS=":"
#shorthand vs explicit variable assignment
1. awk -f script.awk var=value
#var processed after the script run
2. awk -v var=value -f script.awk
#var processed before script execution (gnu/linux only)
3. awk 'BEGIN { name="Sam" }; . . .'
#sets name variable in the BEGIN block before processing input
```

Example:

```
⌜
   BEGIN { count = 0; }
   /\/core$/{
    if ( match( $0, arg1 ) )
     {
      print "rm", $0;
      count++;
     }}
    END { print "echo removed", count, "files"}
                                                  ⌟
```

Match also sets the variables RSTART and RLENGTH
In awk indexing starts at 1 (same with the split() function)
RSTART=1
If failed: RSTART=0, RLENGTH=-1

match() is nearly identical to ($0 ~ arg1), however, the difference is if a match is to be made; it must occur at the beginning of the target string. Similar to $0 ~ "^" (anchor indicates the match would have to be at the beginning of the line.)

```
$ find ~ -name "*core*" -type f | awk -f example9.awk -v 'arg1=/home/ssilvestro/'
echo removed 0 files

#after creating 3 fakes and running the above again
rm /home/ etc . . . #3 times
echo removed 3 files
```

You may have backups located like "/var/backup/home/. . ./", since you don't want to target your backups- theoretically (in this case): "/home/user/" needs to be at the beginning of the match.

#professor reccommends GNU awk manual - built-in functions

Special Variables

```
FS   -> input field separator (defaulted to white space)
OFS  -> output field separator (defaulted to blank)

RS   -> input record separator (defaulted to \n)
ORS  -> output record separator (defaulted to \n)

NF   -> number of fields for the current line
NR   -> record number of the current line
```

Some built-in functions

`int(val)`
returns the truncated int value

`length(val)`
returns the length of the value

`index(str,match)`
returns the index of match in str or 0 if it isn't found

`substr(str,pos,length)`
returns the substring of str beginning at pos for length characters

`split(string, array [, fieldsep])`
splits string into array (indexed at 1) according to fieldsep

When using built-in functions, the expressions that create the function's actual parameters are evaluated completely before the call is performed. For example:

```
i = 4
j = sqrt(i++)
```

i is incremented to the value 5 before sqrt() is called with a value of 4 for its actual parameter.
Avoid writing programs that assume that parameters are evaluated from left to right or from right to left. Another example:

```
i = 5
j = atan2(++i, i *= 2)
```

If the order of evaluation is left to right, i first becomes 6, then 12, then atan2() is called with the two arguments 6 and 12. But if it's right to left: i first becomes 10, then 11, and atan2() is called with the two arguments 10 and 11.

#specific to gawk
mkbool(*expression*) -> returns a boolean-typed value based on the regular boolean value of *expression*. True is 1, false is 0.

#numeric functions

`atan2(y, x)`
Returns the arctangent of y/x in radians.

`pi = atan2(0, -1)`
Retrieves the value of pi

`cos(x)/sin(x)`
Returns cosine/sine of x (in radians)

`exp(x)`
Returns the exponential of x(e^x) or reports an error if x is out of range. Range of values depends on the machine's floating-point rep.

`int(x)`
Returns the nearest integer to x (located between x and 0 and truncated toward 0).
Ex: int(3)=3, int(3.9)=3, int(-3.9)=-3, and int(-3)=-3

`log(x)`
Returns the natural log of x, if x is positive; otherwise returns NaN on IEEE 754 sys.

`rand()`
Returns a random number; values are uniformly distributed between 0 and 1. The value could be 0, but is never 1.

```
      function randint(n)
      {
       return int(n*rand())
      } #used to obtain a random non negative int < n
        #the multiplication produces a rand. number >= 0 && < n
        #using int() this is made into an int between 0 and n-1,
        #inclusive.
```

`sqrt(x)`
Returns positive square root of x #gawk prints warning if x < 0

srand([x])
Set the starting point/seed for generating random numbers to the value of x. If only 'srand()' then the current date and time of day are used for a seed (more unpredictable). The return value of srand() is the previous seed. POSIX doesn't specify initial seed; differs among awk implementations.
#computer-generated numbers are not truly random, only pseudorandom.

#string manipulation functions
([]) -> the parameters are optional
'#' -> gawk-specific
gawk understands locales & does all string processing in terms of characters, not bytes.

asort(*source*[,*dest*[,how]]) #
asorti(*source*[,*dest*[,how]]) #
Both return the number of elements in the array source. For asort(), gawk sorts the values of *source* and replaces the indices of the sorted values with sequential integers (starting with 1). If (optional) array *dest* is specified, then *source* is duplicated into *dest* -> *dest* is sorted -> leaves indices of *source* unchanged.

asorti() is similar, however the indices are sorted instead of the values.

gensub(*regexp*, *replacement*, *how*[, *target*]) #
searches the *target* string target for matches of the regular exp. *regexp*. If *how* is a string beginning with gG (global) -> replace all amtches of *regexp* with *replacement*; otherwise treat *how* as a number indicating which match of *regexp* to replace. If no *target* is supplied, use $0. The original target string isn't changed.

gsub(*regexp*, *replacement* [, *target*])
Searches *target* for all of the longest, leftmost, nonoverlapping matching substrings it can find and replace them with *replacement*. The 'g' stands for global -> replace everywhere.

index(*in*, *find*)
Searches the string *in* for the first occurrence of the string *find*, and returns the position in characters where that occurrence begins in '*in*'.

`length([string])`
Returns the number of characters in a string. If no argument is given, returns the length of $0.

match(*string*, *regexp*[, *array*])
Searches *string* for the longest, leftmost substring matched by the regex *regexp* and returns the character position index at which that substring begins. If no match is found, returns 0.

patsplit(*string*, *array*[, *fieldpat*[, *seps*]]) #
split(*string*, *array*[, *fieldsep*[, seps]])
Divide *string* into fields defined by *fieldpat* and store the pieces in array and the separator strings in the *seps* array.
split() matches the default field split using FS, patsplit() matches the field pattern split.

sprintf(format, expression1, . . .)
Allows you to create strings with specified formats, similar to printf(), but instead of printing to the standard output, it stores the resulting string in a character array provided by the user.

strtonum(*str*) #
Examine *str* and return its numeric value (if beginning with a leading '0', function assumes that *str* is an octal number. If 0x0X then hex.)

sub(*regexp*, *replacement*[, *target*])
Search *target* (treated as a string) for the leftmost, longest substring matched by the regex *regexp*. Modifies the entire string by replacing the matched text with *replacement* -> Modified string becomes new value of *target*. Returns the number of substitutions made.

substr(*string*, *start*[, length])
Returns a length-character-long substring of *string*, starting at character number *start*. The first character of a string is char number one. For example: substr("washington", 5, 3) returns "ing"
Length present, <= 0 -> Returns null string
Length not present -> returns whole suffix of the string
Start < 1 -> Treated as if it was one.
Start > 1 -> Returns null
The string returned by substr() can't be assigned.

`tolower(string)`
Returns a copy of the string with each uppercase character replaced with its corresponding lowercase letter.

`toupper(string)`
Same as above, but lowercase to uppercase.

#Input/Output Functions

`close(filename[,how])`
Close for I/O, second argument is a gawk extension.

`fflush([filename])`
Flush any buffered output associated with the filename, which is either a file opened for writing or a shell command for redirecting output to a pipe or coprocess. If no argument is provided, or its a null string, then awk flushes the buffers for all open output files and pipes.
Successful flush -> returns 0

`system(command)`
Allows the execution of OS commands from within an awk script
#recommend looking at manual for all the details regarding this

#Time Functions

mktime(*dataspec*[, *utc-flag*])
Turns datespec into a timestamp in the same form as is returned by systime().
Form: "*YYYY MM DD HH MM SS* [*DST*]"

strftime([*format*[,*timestamp*[,*utc-flag*]]])
Format the time specified by *timestamp* based on the contents of the *format* string and return the result.
utc-flag present && !0 || !NULL -> formatted as UTC
Else -> formatted for local time zone
timestamp = systime() format

`systime()`
Return the current time as the number of seconds since the system epoch (1970-01-01 00:00:00 UTC on POSIX systems-not counting leap seconds).

```
systime()  -> allows comparison of a timestamp from a log file with the
              current time of day #good to produce log records
mktime()   -> allows conversion of a textual rep. of a date and time
              into a timestamp. #good for before & after comparisons
strftime() -> turn a timestamp into human-readable information.
```

strftime supports the date format specs:

```
%a -> locale weekday name (abbreviated)
%A -> full weekday name
%b -> abbr. month name
%B -> full month name
%c -> locale appropriate date/time rep.
%C -> century part of the current year
%d -> day of the month as a decimal num.
%D -> equivalent to specifying '%m/%d/%y'
%e -> day of the month (padded with space if only 1 digit)
%F -> equivalent to specifying '%Y-%m-%d'
%g -> year modulo 100 of the ISO 8601 week num. #girl what
%G -> The full year of the ISO week number as a decimal number
%h -> equivalent to %b
%H -> 24-hour clock as a decimal number
%I -> 12-hour clock as a decimal number
%j -> day of the year as a decimal number
%m -> month as a decimal number
%M -> minute as a decimal number
%n -> newline
%p -> AM/PM
%r -> locale 12-hour time clock
#there's a million more. . .
```

Example:

```
#include <time.h>
#include <locale.h>
#include <langinfo.h>

struct tm *tm;
char datestring[256];
setlocale (LC_ALL, "");
strftime (datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm);
```

or. . .

```
$ date '+Today is %A, %B %d, %Y.'
Today is Monday, September 22, 2014.
```

#Bit-Manipulation Functions

```
               Bit operator
          |  AND  |   OR  |  XOR
          |---+---+---+---+---+---
Operands  | 0 | 1 | 0 | 1 | 0 | 1
----------+---+---+---+---+---+---
    0     | 0   0 | 0   1 | 0   1
    1     | 0   1 | 1   1 | 1   0
```

`and(v1, v2[,...])`
Return bitwise AND for the arguments (there must be at least two).

`compl(`*val*`)`
Return the bitwise complement of *val*.

`lshift(`*val*`, `*count*`)`
Return the value of *val*, shifted left by *count* bits.

`or(v1, v2[,...])`
Return the bitwise OR of the arguments (there must be at least two).

`rshift(`*val*`, `*count*`)`
Return the value of *val*, shifted right by *count* bits.

`xor(v1, v2[,...])`
Return the bitwise XOR of the arguments (there must be at least two).

#no negatives

#Getting Type Information

`isarray(`*x*`)`
Returns a true value if *x* is an array, otherwise false.

`typeof(`*x*`)`
Return one of the following strings, depending on the type of *x*:
"array", "regexp", "number", "number|bool", "string", "strnum",
"unassigned", "untyped".

#String-Translation Functions

`bindtextdomain(directory[,domain])`
Sets the directory in which gawk will look for message translation files in case they won't/can't be placed in the "standard" locations. It returns the directory in which *domain* is "bound".

`dcgettext(`*string* `[,` *domain* `[,` *category*`] ])`
Returns the translation of *string* in text domain *domain* for locale category *category*.

`dcngettext(`*string1*`,` *string2*`,` *number* `[,` *domain* `[,` *category*`] ])`
Returns the plural form used for *number* of the translation of *string1* and *string2* in text domain *domain* for locale category *category*.

#Various uncategorized examples and notes

FS/RS are field/record separators
By default the input field separator is white space
FS = "[\t]+"
RS = "\n" #new line by default

```
$ cat addresses.txt
SAM SILVESTRO
1 UTSA CIRCLE
SAN ANTONIO, TX 78249

JOHN DOE
123 MAIN STREET
HOUSTON, TX 76565

JANE ROE
555 FIRST STREET
DALLAS, TX 77990
```

#First 3 lines are a record
field 1 = line 1
field 2 = line 2
field 3 = line 3

#What if we wanted to parse this:

```
┌
  awk ‘{ l=1; printf “%d: $s\n”, l, $l
  BEGIN{count = 1}
  {
    print “Address:,” count
    for(l = 1; l <= NF; l++)
     {
      printf “$d: $s\n”, l, $l
     }
      printf(“\n”)
      count++
   }
                                                   ┘
```

#This is screwed up because we didn't change the FS and RS

```
Output:
Address: 9
1: JANE
2: ROE

Address: 10
1: 555
2: FIRST
3: STREET

Address: 11
1: DALLAS,
2: TX
3: 77990
```

#Fixed:

```
┌
  BEGIN
  {
    FS = "\n"
    RS = "" #represents empty lines
    count = 1
  }{
    print “Address:,” count
    for(l = 1; l <= NF; l++)
     {
      printf “$d: $s\n”, l, $l
     }
      printf(“\n”)
      count++
   }
                                                   ┘
```

Output:
```
Address: 1
1: SAM SILVESTRO
2: 1 UTSA CIRCLE
3: SAN ANTONIO, TX 78249

Address: 2
1: JOHN DOE
2: 123 MAIN STREET
3: HOUSTON, TX 76565

Address: 3
1: JANE ROE
2: 555 FIRST STREET
3: DALLAS, TX 77990
```

#ORS is specifically used for the print function

```
$ awk 'BEGIN { print "Name:", "Sam" }'
Name: Sam
$ awk -v OFS=" xxx "
Name: xxx Sam #OFS replaced with space
$ awk 'BEGIN { print "Name:", "Sam" ORS }'
Name: Sam
           #newline
           #newline
$ awk -v ORS='\\n'
Name: Sam\n
$ awk -v ORS='XYZ'
Name: SamXYZ
```

Other examples from sites:

```
#if/else-if statements

awk 'BEGIN{
  test=100;
  if(test>90) {print "very good"}
  else if(test>60) {print "good"}
  else {print "no pass"}
}'
#output: very good
```

Loop Statements

```
#while loop
awk 'BEGIN{
  test=100;
  total=0;
  while(i<=test)
  {
      total+=i;
      i++;
  }
  print total;
}'
#output woud be 5050
```

- - -

```
#for loop
awk 'BEGIN{
  for(k in ENVIRON)
  {
      print k"="ENVIRON[k];
  }
}'
#Output:
TERM=linux
G_BROKEN_FILENAMES=1
SHLVL=1
EDITOR=vim
PWD=/root
TMOUT=6000
HISTTIMEFORMAT=%F %T --
...........
```

```
#do-while
awk 'BEGIN{
  total=0;
  i=0;
  do
  {
      total+=i;
      i++;
  }
  while(i<=100)
  print total;
}'
#output is 5050
```

Built-in variables

```
$0 -> current record
$1~$n -> The nth field of the current record separated by FS

$ awk '/^root/{print $0}' /etc/passwd
root:x:0:0:root:/root:/bin/bash

$ awk 'BEGIN{FS=":"}/^root/{print $1,$NF}' /etc/passwd
root /bin/bash

$ awk 'BEGIN{FS=":"}{print NR,$1,$NF}' /etc/passwd
1 root /bin/bash
2 lp /sbin/nologin
3 sync /bin/sync
etc . . #$NF refers to the last column

#More built-in variables
ORS -> output record separator (default: newline)
OFS -> output field separator (default: space)
ARGC -> num. of command-line arguments
ARGV -> array of command-line arguments.

$ awk 'BEGIN{FS=":";ORS="**"}{print FNR,$1,$NF}' /etc/passwd
1 root /bin/bash**2 lp /sbin/nologin**3 sync /bin/sync**4 shutdown
/sbin/shutdown**5 halt etc . . .

$ awk 'BEGIN{FS=":";print "ARGC="ARGC;for(k in ARGV) {print k"="ARGV[k]; }}' /etc/passwd
ARGC=2
0=awk
1=/etc/passwd
```

Math

```
$ awk 'BEGIN{a=int(8.998273485);print a}'
8
```

Strings

```
$ awk 'BEGIN{info="this is a test123test456!";sub(/[0-9]+/,"!",info);print info}'
this is a test!test456!

$ awk 'BEGIN{info="this is a test123test456!";gsub(/[0-9]+/,"!",info);print info}'
this is a test!test!!
```

```
$ awk 'BEGIN{info="this is a test123test!!";print
index(info,"test")?"test exist!":"no found test!"}'
test exist!
```

```
$ awk 'BEGIN{info="this is a test123test!";print
match(info,/[0-9]+/)?"num exist":"no found num!"}'
num exist
```

```
$ awk 'BEGIN{info="1234567abc";print substr(info,4,6)}'
4567ab
```

```
$ awk 'BEGIN{info="this is a test";split(info,a," ");for(i in a);print i,a}'
3 a
```

```
$ awk 'BEGIN{info="this is a test haha";split(info,ali," ");print
length(ali);for(k in ali){print k,ali[k]}}'
5
4 test
5 haha
1 this
2 is
3 a
```

General

```
$ awk 'BEGIN{print "input your name:";getline name;print name}'
input your name:
joe
mama
```

```
$ awk 'BEGIN{b=system("date;cal");print b}'
Fri Apr 11 22:39:21 CDT 2025
      April 2025
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

Array related functions

```
$ awk 'BEGIN{info="this is a test";len=split(info,a," ");print
length(a),length(info),len}'
4 14 4
```

```
$ awk 'BEGIN{info="this is a test";split(info,a," ");for(i in a){print
i,a}}'
4 test
1 this
2 is
3 a
```

```
$ awk 'BEGIN{info="this is a test";len=split(info,a,"
");for(i=1;i<=len;i++){print i,a}}'
1 this
2 is
3 a
4 test
```

```
$ awk 'BEGIN{a[1]="b";c["d"]="f";if(a[1]=="b"){print
"good"};if(c["d"]!="g"){print "good again"}}'
good
good again
```

Sorting arrays

```
awk 'BEGIN{
  a[100]=100;
  a[2]=224;
  a[3]=45;
  slen=asort(a,b); #asort sorts array values, discard og keys
  for(i=1;i<=slen;i++)
  {print i,b[i]}
}'
#Output:
1 45
2 100
3 224
- - -
awk 'BEGIN{
  a["f"]=100;
  a["a"]=893;
  a["b"]=90;
  slen=asorti(a,b); #asorti sorts keys and stores them in array b
  for(i=1;i<=slen;i++)
  {print i,b[i],a[b[i]]}
}'
#Output:
1 a 893
2 b 90
3 f 100
```

```
#Random test
#BEGIN { } -> Statements executed before processing input
#END { } -> Statements executed after processing all input
#{ } -> Statements executed for each line of input

#using BEGIN/END directly in awk command
$ awk 'BEGIN{printf "%4s %4s %4s %4s %4s %4s %4s %4s %4s\n",
"FILENAME","ARGC","FNR","FS","NS","NF","NR","OFS","ORS","RS";
printf"----------..etc\n"} {printf "%4s %4s %4s %4s %4s %4s %4s %4s
%4s\n",FILENAME,ARGC,FNR,FS,NF,NR,OFS,ORS,RS}' log.txt
FILENAME ARGC FNR FS NF NR OFS ORS RS
-------------------------------------
log.txt    2   1     5  1
log.txt    2   2     5  2
log.txt    2   3     3  3
and so on. . .

#standalone awk script
$ cat score.txt
Marry   2143 78 84 77
Jack    2321 66 78 45
Tom     2122 48 77 71
Mike    2537 87 97 95
Bob     2415 40 57 62
BEGIN
{
    math = 0
    english = 0
    computer = 0

    printf "NAME    NO.   MATH  ENGLISH  COMPUTER   TOTAL\n"
    printf "---------------------------------------------\n"
}
#processing each line
{
    math+=$3
    english+=$4
    computer+=$5
    printf "%-6s %-6s %4d %8d %8d %8d\n", $1, $2, $3,$4,$5, $3+$4+$5
}
#after processing
END
{
    printf "---------------------------------------------\n"
    printf "  TOTAL:%10d %8d %8d \n", math, english, computer
    printf "AVERAGE:%10.2f %8.2f %8.2f\n", math/NR, english/NR, computer/NR
}
```

```
$ awk -f cal.awk score.txt
NAME    NO.   MATH  ENGLISH  COMPUTER   TOTAL
---------------------------------------------
Marry  2143     78       84        77     239
Jack   2321     66       78        45     189
Tom    2122     48       77        71     196
Mike   2537     87       97        95     279
Bob    2415     40       57        62     159
---------------------------------------------
  TOTAL:       319      393       350
AVERAGE:     63.80    78.60     70.00
```