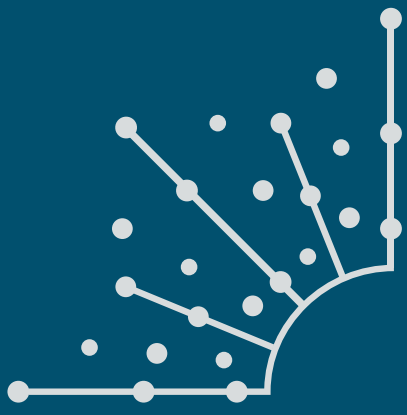
A practical guide to learning GNU Awk



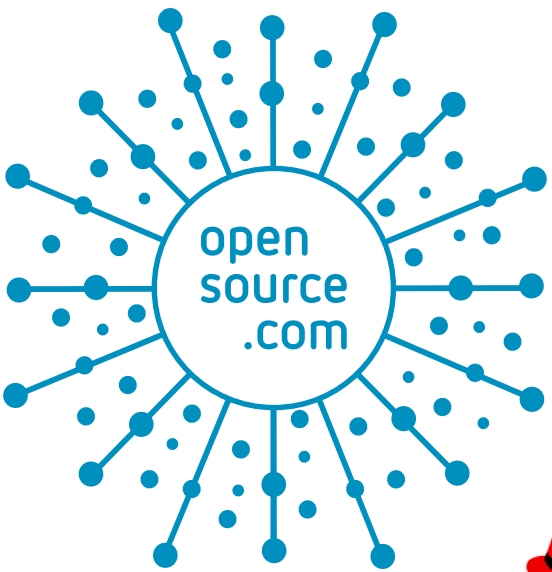
ABOUT OPENSOURCE.COM

What is Opensource.com?

OPENSOURCE.COM publishes stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Submit a story idea: opensource.com/story

Email us: open@opensource.com



Supported by
Red Hat

AUTHORS SETH KENLON, DAVE MORRISS, AND ROBERT YOUNG

SETH KENLON is an independent multimedia artist, free culture advocate, and UNIX geek. He has worked in the film and computing industry, often at the same time. He is one of the maintainers of the Slackware-based multimedia production project, http://slackermedia.info.

DAVE MORRISS is a retired IT Manager now contributing to the "Hacker Public Radio" community podcast (http://hackerpublicradio.org) as a podcast host and an administrator.

ROBERT YOUNG is the Owner and Principal Consultant at Lab Insights, LLC. He has led dozens of laboratory informatics and data manage projects over the last 10 years. Robert Holds a degree in Cell Biology/Biochemistry and a masters in Bioinformatics.

CONTRIBUTORS

Jim Hall

Lazarus Lazaridis

Dave Neary

Moshe Zadka

CONTENTS

CHAPTERS

LEARN

PRACTICE

CHEAT SHEET

What is awk?

awk is known for its robust ability to process and interpret data from text files.

AWK is a programming language and a POSIX [1] specification that originated at AT&T Bell Laboratories in 1977. Its name comes from the initials of its designers: Aho, Weinberger, and Kernighan. awk features user-defined functions, multiple input streams, TCP/IP networking access, and a rich set of regular expressions. It's often used to process raw text files, interpreting the data it finds as records and fields to be manipulated by the user.

At its most basic, awk searches files for some unit of text (usually lines terminated with an end-of-line character) containing some user-specified pattern. When a line matches one of the patterns, awk performs some set of user-defined actions on that line, then processes input lines until the end of the input files.

awk is used as a command as often as it is used as an interpreted script. One-liners are popular and useful ways of filtering output from files or output streams or as standalone commands. awk even has an interactive mode of sorts because, without input, it acts upon any line the user types into the terminal:

```
$ awk '/foo/ { print toupper($0); }'
This line contains bar.
This line contains foo.
THIS LINE CONTAINS FOO.
```

However, awk is a programming language with user-defined functions, loops, conditionals, flow control, and more. It's robust enough as a language that it has been used to program a wiki and even (believe it or not) a retargetable assembler for eight-bit microprocessors.

Why use awk?

awk may seem outdated in a world fortunate enough to have Python available by default on several major operating systems, but its longevity is well-earned. In many ways, programs written in awk are different from programs in other languages because awk is data-driven. That is, you describe to awk what data you want to work with and then what you want it to do when such data is found. There are no boilerplate constructors to create, no elaborate class structure to design, no stream objects to create. awk is built for a specific purpose, so there's a lot you can take for granted and allow awk to handle.

What's the difference between awk and gawk?

Awk is an open source POSIX specification, so anyone can (in theory) implement a version of the command and language. On Linux or any system that provides GNU awk [2], the command to invoke awk is **gawk**, but it's symlinked to the generic command **awk**. The same is true for systems that provide **nawk** or **mawk** or any other variety of awk implementation. Most versions of awk implement the core functionality and literal functions defined by the POSIX spec, although they may add special new features not present in others. For that reason, there's some risk of learning one implementation and coming to rely on a special feature, but this "problem" is tempered by the fact that most of them are open source, so they usually can be installed as needed.

Learning awk

There are many great resources for learning awk. The GNU awk manual, *GAWK: Effective awk programming* [3], is a definitive guide to the language. You can find many other tutorials for awk [4] on Opensource.com, including "Getting started with awk, a powerful text-parsing tool." [5]

Links

[1] https://opensource.com/article/19/7/what-posix-richard-stallman-explains
[2] https://www.gnu.org/software/gawk/
[3] https://www.gnu.org/software/gawk/manual/
[4] https://opensource.com/sitewide-search?search_api_views_fulltext=awk
[5] https://opensource.com/article/19/10/intro-awk

Getting started with awk, a powerful text-parsing tool

Let's jump in and start using it.

AWK IS A POWERFUL text-parsing tool for Unix and Unix-like systems, but because it has programmed functions that you can use to perform common parsing tasks, it's also considered a programming language. You probably won't be developing your next GUI application with awk, and it likely won't take the place of your default scripting language, but it's a powerful utility for specific tasks.

What those tasks may be is surprisingly diverse. The best way to discover which of your problems might be best solved by awk is to learn awk; you'll be surprised at how awk can help you get more done but with a lot less effort.

Awk's basic syntax is:

```
awk [options] 'pattern {action}' file
```

To get started, create this sample file and save it as **colours.txt**

```
name       color  amount
apple      red    4
banana     yellow 6
strawberry red    3
grape      purple 10
apple      green  8
plum       purple 2
kiwi       brown  4
potato     brown  9
pineapple  yellow 5
```

This data is separated into columns by one or more spaces. It's common for data that you are analyzing to be organized in some way. It may not always be columns separated by whitespace, or even a comma or semicolon, but especially in log files or data dumps, there's generally a predictable pattern. You can use patterns of data to help awk extract and process the data that you want to focus on.

Printing a column

In awk, the **print** function displays whatever you specify. There are many predefined variables you can use, but some of the most common are integers designating columns in a text file. Try it out:

```
$ awk '{print $2;}' colours.txt
color
red
yellow
red
purple
green
purple
brown
brown
yellow
```

In this case, awk displays the second column, denoted by **$2**. This is relatively intuitive, so you can probably guess that **print $1** displays the first column, and **print $3** displays the third, and so on.

To display *all* columns, use **$0**.

The number after the dollar sign (**$**) is an expression, so **$2** and **$(1+1)** mean the same thing.

Conditionally selecting columns

The example file you're using is very structured. It has a row that serves as a header, and the columns relate directly to one another. By defining conditional requirements, you can qualify what you want awk to return when looking at this data. For instance, to view items in column 2 that match "yellow" and print the contents of column 1:

```
awk '$2=="yellow"{print $1}' colours.txt
banana
pineapple
```

Regular expressions work as well. This conditional looks at **$2** for approximate matches to the letter **p** followed by any number of (one or more) characters, which are in turn followed by the letter **p**:

```
$ awk '$2 ~ /p.+p/ {print $0}' colours.txt
grape   purple  10
plum    purple  2
```

Numbers are interpreted naturally by awk. For instance, to print any row with a third column containing an integer greater than 5:

```
awk '$3>5 {print $1, $2}' colours.txt
name    color
banana  yellow
grape   purple
apple   green
potato  brown
```

Field separator

By default, awk uses whitespace as the field separator. Not all text files use whitespace to define fields, though. For example, create a file called **colours.csv** with this content:

```
name,color,amount
apple,red,4
banana,yellow,6
strawberry,red,3
grape,purple,10
apple,green,8
plum,purple,2
kiwi,brown,4
potato,brown,9
pineapple,yellow,5
```

Awk can treat the data in exactly the same way, as long as you specify which character it should use as the field separator in your command. Use the **--field-separator** (or just **-F** for short) option to define the delimiter:

```
$ awk -F"," '$2=="yellow" {print $1}' file1.csv
banana
pineapple
Saving output
```

Using output redirection, you can write your results to a file. For example:

```
$ awk -F, '$3>5 {print $1, $2} colours.csv > output.txt
```

This creates a file with the contents of your awk query.

You can also split a file into multiple files grouped by column data. For example, if you want to split colours.txt into multiple files according to what color appears in each row, you can cause awk to redirect *per query* by including the redirection in your awk statement:

```
$ awk '{print > $2".txt"}' colours.txt
```

This produces files named **yellow.txt**, **red.txt**, and so on.

Fields, records, and variables in awk

In the second article in this intro to awk series, learn about fields, records, and some powerful awk variables.

AWK COMES in several varieties: There is the original **awk**, written in 1977 at AT&T Bell Laboratories, and several reimplementations, such as **mawk**, **nawk**, and the one that ships with most Linux distributions, GNU awk, or **gawk**. On most Linux distributions, awk and gawk are synonyms referring to GNU awk, and typing either invokes the same awk command. See the GNU awk user's guide [1] for the full history of awk and gawk.

The first article in this series showed that awk is invoked on the command line with this syntax:

```
$ awk [options] 'pattern {action}' inputfile
```

Awk is the command, and it can take options (such as **-F** to define the field separator). The action you want awk to perform is contained in single quotes, at least when it's issued in a terminal. To further emphasize which part of the awk command is the action you want it to take, you can precede your program with the **-e** option (but it's not required):

```
$ awk -F, -e '{print $2;}' colours.txt
yellow
blue
green
[...]
```

Records and fields

Awk views its input data as a series of *records*, which are usually newline-delimited lines. In other words, awk generally sees each line in a text file as a new record. Each record contains a series of *fields*. A field is a component of a record delimited by a *field separator*.

By default, awk sees whitespace, such as spaces, tabs, and newlines, as indicators of a new field. Specifically, awk treats multiple space separators as one, so this line contains two fields:

```
raspberry red
```

As does this one:

```
tuxedo                  black
```

Other separators are not treated this way. Assuming that the field separator is a comma, the following example record contains three fields, with one probably being zero characters long (assuming a non-printable character isn't hiding in that field):

```
a,,b
```

The awk program

The *program* part of an awk command consists of a series of rules. Normally, each rule begins on a new line in the program (although this is not mandatory). Each rule consists of a pattern and one or more actions:

```
pattern { action }
```

In a rule, you can define a pattern as a condition to control whether the action will run on a record. Patterns can be simple comparisons, regular expressions, combinations of the two, and more.

For instance, this will print a record *only* if it contains the word "raspberry":

```
$ awk '/raspberry/ { print $0 }' colours.txt
raspberry red 99
```

If there is no qualifying pattern, the action is applied to every record.

Also, a rule can consist of only a pattern, in which case the entire record is written as if the action was **{ print }**.

Awk programs are essentially *data-driven* in that actions depend on the data, so they are quite a bit different from programs in many other programming languages.

The NF variable

Each field has a variable as a designation, but there are special variables for fields and records, too. The variable **NF** stores the number of fields awk finds in the current record. This can be printed or used in tests. Here is an example using the text file [2] from the previous article:

```
$ awk '{ print $0 " (" NF ")" }' colours.txt
name       color  amount (3)
apple      red    4 (3)
banana     yellow 6 (3)
[...]
```

Awk's **print** function takes a series of arguments (which may be variables or strings) and concatenates them together. This is why, at the end of each line in this example, awk prints the number of fields as an integer enclosed by parentheses.

The NR variable

In addition to counting the fields in each record, awk also counts input records. The record number is held in the variable **NR**, and it can be used in the same way as any other variable. For example, to print the record number before each line:

```
$ awk '{ print NR ": " $0 }' colours.txt
1: name       color  amount
2: apple      red    4
3: banana     yellow 6
4: raspberry  red    3
5: grape      purple 10
[...]
```

Note that it's acceptable to write this command with no spaces other than the one after **print**, although it's more difficult for a human to parse:

```
$ awk '{print NR": "$0}' colours.txt
```

The printf() function

For greater flexibility in how the output is formatted, you can use the awk **printf()** function. This is similar to **printf** in C, Lua, Bash, and other languages. It takes a *format* argument followed by a comma-separated list of items. The argument list may be enclosed in parentheses.

```
$ printf format, item1, item2, ...
```

The format argument (or *format string*) defines how each of the other arguments will be output. It uses *format specifiers* to do this, including **%s** to output a string and **%d** to output a decimal number. The following **printf** statement outputs the record followed by the number of fields in parentheses:

```
$ awk 'printf "%s (%d)\n",$0,NF}' colours.txt
name       color  amount (3)
raspberry  red    4 (3)
banana     yellow 6 (3)
[...]
```

In this example, **%s (%d)** provides the structure for each line, while **$0,NF** defines the data to be inserted into the **%s** and **%d** positions. Note that, unlike with the **print** function, no newline is generated without explicit instructions. The escape sequence **\n** does this.

Awk scripting

All of the awk code in this article has been written and executed in an interactive Bash prompt. For more complex programs, it's often easier to place your commands into a file or script. The option **-f FILE** (not to be confused with **-F**, which denotes the field separator) may be used to invoke a file containing a program.

For example, here is a simple awk script. Create a file called **example1.awk** with this content:

```
/^a/ {print "A: " $0}
/^b/ {print "B: " $0}
```

It's conventional to give such files the extension **.awk** to make it clear that they hold an awk program. This naming is not mandatory, but it gives file managers and editors (and you) a useful clue about what the file is.

Run the script:

```
$ awk -f example1.awk colours.txt
A: raspberry  red    4
B: banana     yellow 6
A: apple      green  8
```

A file containing awk instructions can be made into a script by adding a **#!** line at the top and making it executable. Create a file called **example2.awk** with these contents:

```
#!/usr/bin/awk -f
#
# Print all but line 1 with the line number on the front
#

NR > 1 {
    printf "%d: %s\n",NR,$0
}
```

Arguably, there's no advantage to having just one line in a script, but sometimes it's easier to execute a script than to remember and type even a single line. A script file also provides a good opportunity to document what a command does. Lines starting with the # symbol are comments, which awk ignores.

Grant the file executable permission:

```
$ chmod u+x example2.awk
```

Run the script:

```
$ ./example2.awk colours.txt
2: apple      red    4
2: banana     yellow 6
4: raspberry  red    3
5: grape      purple 10
[...]
```

An advantage of placing your awk instructions in a script file is that it's easier to format and edit. While you can write awk on a single line in your terminal, it can get overwhelming when it spans several lines.

Try it

You now know enough about how awk processes your instructions to be able to write a complex awk program. Try writing an awk script with more than one rule and at least one conditional pattern. If you want to try more functions than just **print** and **printf**, refer to the gawk manual [3] online.

Here's an idea to get you started:

```
#!/usr/bin/awk -f
#
# Print each record EXCEPT
# IF the first record contains "raspberry",
# THEN replace "red" with "pi"

$1 == "raspberry" {
        gsub(/red/,"pi")
}

{ print }
```

Try this script to see what it does, and then try to write your own.

Links

[1] https://www.gnu.org/software/gawk/manual/html_node/History.html#History
[2] https://opensource.com/article/19/10/intro-awk
[3] https://www.gnu.org/software/gawk/manual/

A guide to intermediate awk scripting

Learn how to structure commands into executable scripts.

THIS ARTICLE explores awk's capabilities, which are easier to use now that you know how to structure your command into an executable script.

Logical operators and conditionals

You can use the logical operators and (written **&&**) and **or** (written ||) to add specificity to your conditionals.

For example, to select and print only records with the string "purple" in the second column *and* an amount less than five in the third column:

```
$2 == "purple" && $3 < 5 {print $1}
```

If a record has "purple" in column two but a value greater than or equal to 5 in column three, then it is *not* selected. Similarly, if a record matches column three's requirement but lacks "purple" in column two, it is also not selected.

Next command

Say you want to select every record in your file where the amount is greater than or equal to eight and print a matching record with two asterisks (**). You also want to flag every record with a value between five (inclusive) and eight with only one asterisk (*). There are a few ways to do this, and one way is to use the **next** command to instruct awk that after it takes an action, it should stop scanning and proceed to the *next* record.

Here's an example:

```
NR == 1 {
  print $0;
  next;
}

$3 >= 8 {
  printf "%s\t%s\n", $0, "**";
  next;
}

$3 >= 5 {
  printf "%s\t%s\n", $0, "*";
  next;
}

$3 < 5 {
  print $0;
}
```

BEGIN command

The **BEGIN** command lets you print and set variables before awk starts scanning a text file. For instance, you can set the input and output field separators inside your awk script by defining them in a **BEGIN** statement. This example adapts the simple script from the previous article for a file with fields delimited by commas instead of whitespace:

```
#!/usr/bin/awk -f
#
# Print each record EXCEPT
# IF the first record contains "raspberry",
# THEN replace "red" with "pi"

BEGIN {
        FS=",";
}

$1 == "raspberry" {
        gsub(/red/,"pi")
}
```

END command

The **END** command, like **BEGIN**, allows you to perform actions in awk after it completes its scan through the text

file you are processing. If you want to print cumulative results of some value in all records, you can do that only after all records have been scanned and processed.

The **BEGIN** and **END** commands run only once each. All rules between them run zero or more times on each *record*. In other words, most of your awk script is a loop that is executed at every new line of the text file you're processing, with the exception of the **BEGIN** and **END** rules, which run before and after the loop.

Here is an example that wouldn't be possible without the **END** command. This script accepts values from the output of the **df** Unix command and increments two custom variables (**used** and **available**) with each new record.

```
$1 != "tempfs" {
    used += $3;
    available += $4;
}

END {
    printf "%d GiB used\n%d GiB available\n",
used/2^20, available/2^20;
}
```

Save the script as **total.awk** and try it:

```
df -l | awk -f total.awk
```

The **used** and **available** variables act like variables in many other programming languages. You create them arbitrarily and without declaring their type, and you add values to them at will. At the end of the loop, the script adds the records in the respective columns together and prints the totals.

Math

As you can probably tell from all the logical operators and casual calculations so far, awk does math quite naturally. This arguably makes it a very useful calculator for your terminal. Instead of struggling to remember the rather unusual syntax of **bc**, you can just use awk along with its special **BEGIN** function to avoid the requirement of a file argument:

```
$ awk 'BEGIN { print 2*21 }'
42
$ awk 'BEGIN {print 8*log(4) }'
11.0904
```

Admittedly, that's still a lot of typing for simple (and not so simple) math, but it wouldn't take much effort to write a frontend, which is an exercise for you to explore.

How to use loops in awk

Learn how to use different types of loops to run commands on a record multiple times.

AWK SCRIPTS have three main sections: the optional BEGIN and END functions and the functions you write that are executed on each record. In a way, the main body of an awk script is a loop, because the commands in the functions run for each record. However, sometimes you want to run commands on a record more than once, and for that to happen, you must write a loop.

There are several kinds of loops, each serving a unique purpose.

While loop

A **while** loop tests a condition and performs commands *while* the test returns *true*. Once a test returns *false*, the loop is broken.

```
#!/bin/awk -f

BEGIN {
        # Loop through 1 to 10

    i=1;
    while (i <= 10) {
        print i, " to the second power is ", i*i;
        i = i+1;
    }
exit;
}
```

In this simple example, awk prints the square of whatever integer is contained in the variable *i*. The **while (i <= 10)** phrase tells awk to perform the loop only as long as the value of *i* is less than or equal to 10. After the final iteration (while *i* is 10), the loop ends.

Do while loop

The **do while** loop performs commands after the keyword **do**. It performs a test afterward to determine whether the stop condition has been met. The commands are repeated only *while* the test returns true (that is, the end condition has *not* been met). If a test fails, the loop is broken because the end condition has been met.

```
#!/usr/bin/awk -f
BEGIN {

        i=2;
        do {
                print i, " to the second power is ", i*i;
                i = i + 1
        }
        while (i < 10)

exit;
}
```

For loops

There are two kinds of **for** loops in awk.

One kind of **for** loop initializes a variable, performs a test, and increments the variable together, performing commands while the test is true.

```
#!/bin/awk -f

BEGIN {
    for (i=1; i <= 10; i++) {
        print i, " to the second power is ", i*i;
    }
exit;
}
```

Another kind of **for** loop sets a variable to successive indices of an array, performing a collection of commands for each index. In other words, it uses an array to "collect" data from a record.

This example implements a simplified version of the Unix command **uniq**. By adding a list of strings into an array called **a** as a key and incrementing the value each time the same key occurs, you get a count of the number of times a string appears (like the **--count** option of **uniq**). If you print the keys of the array, you get every string that appears one or more times.

For example, using the demo file **colours.txt** (from the previous articles):

```
name       color  amount
apple      red    4
banana     yellow 6
raspberry  red    99
strawberry red    3
grape      purple 10
apple      green  8
plum       purple 2
kiwi       brown  4
potato     brown  9
pineapple  yellow 5
```

Here is a simple version of **uniq -c** in awk form:

```
#! /usr/bin/awk -f

NR != 1 {
    a[$2]++
}
END {
    for (key in a) {
                print a[key] " " key
    }
}
```

The third column of the sample data file contains the number of items listed in the first column. You can use an array and a **for** loop to tally the items in the third column by color:

```
#! /usr/bin/awk -f

BEGIN {
    FS=" ";
    OFS="\t";
    print("color\tsum");
}
NR != 1 {
    a[$2]+=$3;
}
END {
    for (b in a) {
        print b, a[b]
    }
}
```

As you can see, you are also printing a header column in the **BEFORE** function (which always happens only once) prior to processing the file.

Loops

Loops are a vital part of any programming language, and awk is no exception. Using loops can help you control how your awk script runs, what information it's able to gather, and how it processes your data. Our next article will cover switch statements, **continue**, and **next**.

How to use regular expressions in awk

Use regex to search code using dynamic and complex pattern definitions.

IN AWK, regular expressions (regex) allow for dynamic and complex pattern definitions. You're not limited to searching for simple strings but also patterns within patterns.

The syntax for using regular expressions to match lines in awk is:

```
word ~ /match/
```

The inverse of that is not matching a pattern:

```
word !~ /match/
```

If you haven't already, create the sample file from our previous article:

```
name       color  amount
apple      red    4
banana     yellow 6
strawberry red    3
raspberry  red    99
grape      purple 10
apple      green  8
plum       purple 2
kiwi       brown  4
potato     brown  9
pineapple  yellow 5
```

Save the file as **colours.txt** and run:

```
$ awk -e '$1 ~ /p[el]/ {print $0}' colours.txt
apple      red    4
grape      purple 10
apple      green  8
plum       purple 2
pineapple  yellow 5
```

You have selected all records containing the letter **p** followed by *either* an **e** or an **l**.

Adding an **o** inside the square brackets creates a new pattern to match:

```
$ awk -e '$1 ~ /p[o]/ {print $0}' colours.txt
apple      red    4
grape      purple 10
apple      green  8
plum       purple 2
pineapple  yellow 5
potato     brown  9
```

Regular expression basics

Certain characters have special meanings when they're used in regular expressions.

Anchors

Anchor	Function
^	Indicates the beginning of the line
$	Indicates the end of a line
\A	Denotes the beginning of a string
\z	Denotes the end of a string
\b	Marks a word boundary

For example, this awk command prints any record containing an **r** character:

```
$ awk -e '$1 ~ /r/ {print $0}' colours.txt
strawberry red    3
raspberry  red    99
grape      purple 10
```

Add a **^** symbol to select only records where **r** occurs at the beginning of the line:

```
$ awk -e '$1 ~ /^r/ {print $0}' colours.txt
raspberry  red    99
```

Characters

Character	Function
[ad]	Selects **a** or **d**
[a-d]	Selects any character **a** through **d** (a, b, c, or d)
[^a-d]	Selects any character *except* **a** through **d** (e, f, g, h…)
\w	Selects any word
\s	Selects any whitespace character
\d	Selects any digit

The capital versions of w, s, and d are negations; for example, **\D** *does not* select any digit.

POSIX [1] regex offers easy mnemonics for character classes:

POSIX mnemonic	Function
[:alnum:]	Alphanumeric characters
[:alpha:]	Alphabetic characters
[:space:]	Space characters (such as space, tab, and formfeed)
[:blank:]	Space and tab characters
[:upper:]	Uppercase alphabetic characters
[:lower:]	Lowercase alphabetic characters
[:digit:]	Numeric characters
[:xdigit:]	Characters that are hexadecimal digits
[:punct:]	Punctuation characters (i.e., characters that are not letters, digits, control characters, or space characters)
[:cntrl:]	Control characters
[:graph:]	Characters that are both printable and visible (e.g., a space is printable but not visible, whereas an **a** is both)
[:print:]	Printable characters (i.e., characters that are not control characters)

Quantifiers

Quantifier	Function
.	Matches any character
+	Modifies the preceding set to mean *one or more times*
*****	Modifies the preceding set to mean *zero or more times*
?	Modifies the preceding set to mean *zero or one time*
{n}	Modifies the preceding set to mean *exactly n times*
{n,}	Modifies the preceding set to mean *n or more times*
{n,m}	Modifies the preceding set to mean *between n and m times*

Many quantifiers modify the character sets that precede them. For example, **.** means any character that appears exactly once, but **.*** means *any* or *no* character. Here's an example; look at the regex pattern carefully:

```
$ printf "red\nrd\n"
red
rd
$ printf "red\nrd\n" | awk -e '$0 ~ /^r.d/ {print}'
red
$ printf "red\nrd\n" | awk -e '$0 ~ /^r.*d/ {print}'
red
rd
```

Similarly, numbers in braces specify the number of times something occurs. To find records in which an **e** character occurs exactly twice:

```
$ awk -e '$2 ~ /e{2}/ {print $0}' colours.txt
apple      green  8
```

Grouped matches

Quantifier	Function
(red)	Parentheses indicate that the enclosed letters must appear contiguously
|	Means or in the context of a grouped match

For instance, the pattern (**red**) matches the word **red** and **ordered** but not any word that contains all three of those letters in another order (such as the word **order**).

Awk like sed with sub() and gsub()

Awk features several functions that perform find-and-replace actions, much like the Unix command **sed**. These are functions, just like **print** and **printf**, and can be used in awk rules to replace strings with a new string, whether the new string is a string or a variable.

The **sub** function substitutes the *first* matched entity (in a record) with a replacement string. For example, if you have this rule in an awk script:

```
{ sub(/apple/, "nut", $1);
    print $1 }
```

running it on the example file **colours.txt** produces this output:

```
name
nut
banana
raspberry
strawberry
grape
nut
```

```
plum
kiwi
potato
pinenut
```

The reason both **apple** and **pineapple** were replaced with **nut** is that both are the first match of their records. If the records were different, then the results could differ:

```
$ printf "apple apple\npineapple apple\n" | \
awk -e 'sub(/apple/, "nut")'
nut apple
pinenut apple
```

The **gsub** command substitutes *all* matching items:

```
$ printf "apple apple\npineapple apple\n" | \
awk -e 'gsub(/apple/, "nut")'
nut nut
pinenut nut
```

Gensub

An even more complex version of these functions, called **gensub()**, is also available.

The **gensub** function allows you to use the **&** character to recall the matched text. For example, if you have a file with the word **Awk** and you want to change it to **GNU Awk**, you could use this rule:

```
{ print gensub(/(Awk)/, "GNU &", 1) }
```

This searches for the group of characters **Awk** and stores it in memory, represented by the special character **&**. Then it substitutes the string for **GNU &**, meaning **GNU Awk**. The **1** character at the end tells **gensub()** to replace the first occurrence.

```
$ printf "Awk\nAwk is not Awkward" \
| awk -e ' { print gensub(/(Awk)/, "GNU &",1) }'
GNU Awk
GNU Awk is not Awkward
```

There's a time and a place

Awk is a powerful tool, and regex are complex. You might think awk is so very powerful that it could easily replace **grep** and **sed** and **tr** and **sort** [2] and many more, and in a sense, you'd be right. However, awk is just one tool in a toolbox that's overflowing with great options. You have a choice about what you use and when you use it, so don't feel that you have to use one tool for every job great and small.

With that said, awk really is a powerful tool with lots of great functions. The more you use it, the better you get to know it. Remember its capabilities, and fall back on it occasionally so can you get comfortable with it.

Links

[1] https://opensource.com/article/19/7/what-posix-richard-stallman-explains
[2] https://opensource.com/article/19/10/get-sorted-sort

4 ways to control the flow of your awk script

Learn to use switch statements and the break, continue, and next commands to control awk scripts.

THERE ARE MANY WAYS to control the flow of an awk script, including loops [1], **switch** statements and the **break**, **continue**, and **next** commands.

Sample data

Create a sample data set called **colours.txt** and copy this content into it:

```
name       color  amount
apple      red    4
banana     yellow 6
strawberry red    3
raspberry  red    99
grape      purple 10
apple      green  8
plum       purple 2
kiwi       brown  4
potato     brown  9
pineapple  yellow 5
Switch statements
```

The **switch** statement is a feature specific to GNU awk, so you can only use it with **gawk**. If your system or your target system doesn't have **gawk**, then you should not use a switch statement.

The **switch** statement in **gawk** is similar to the one in C and many other languages. The syntax is:

```
switch (expression) {
        case VALUE:
                <do something here>
        [...]
        default:
                <do something here>
}
```

The **expression** part can be any awk expression that returns a numeric or string result. The **VALUE** part (after the word **case**) is a numeric or string constant or a regular expression.

When a **switch** statement runs, the *expression* is evaluated, and the result is matched against each case value. If there's a match, then the code contained within a case definition is executed. If there's no match in any case definition, then the default statement is executed.

The keyword **break** is at the end of the code in each case definition to break the loop. Without **break**, awk would continue to search for matching case values.

Here's an example **switch** statement:

```
#!/usr/bin/awk -f
#
# Example of the use of 'switch' in GNU Awk.

NR > 1 {
    printf "The %s is classified as: ",$1

    switch ($1) {
        case "apple":
            print "a fruit, pome"
            break
        case "banana":
        case "grape":
        case "kiwi":
            print "a fruit, berry"
            break
                case "raspberry":
                        print "a computer, pi"
                        break
        case "plum":
            print "a fruit, drupe"
            break
```

```
        case "pineapple":
            print "a fruit, fused berries (syncarp)"
            break
        case "potato":
            print "a vegetable, tuber"
            break
        default:
            print "[unclassified]"
    }
}
```

This script notably ignores the first line of the file, which in the case of the sample data is just a header. It does this by operating only on records with an index number greater than 1. On all other records, this script compares the contents of the first field (**$1**, as you know from previous articles) to the value of each **case** definition. If there's a match, the print function is used to **print** the botanical classification of the entry. If there are no matches, then the **default** instance prints **"[unclassified]"**.

The banana, grape, and kiwi are all botanically classified as a berry, so there are three **case** definitions associated with one print result.

Run the script on the **colours.txt** sample file, and you should get this:

```
The apple is classified as: a fruit, pome
The banana is classified as: a fruit, berry
The strawberry is classified as: [unclassified]
The raspberry is classified as: a computer, pi
The grape is classified as: a fruit, berry
The apple is classified as: a fruit, pome
The plum is classified as: a fruit, drupe
The kiwi is classified as: a fruit, berry
The potato is classified as: a vegetable, tuber
The pineapple is classified as: a fruit, fused berries
    (syncarp)
```

Break

The **break** statement is mainly used for the early termination of a **for**, **while**, or **do-while** loop or a **switch** statement. In a loop, **break** is often used where it's not possible to determine the number of iterations of the loop beforehand. Invoking **break** terminates the enclosing loop (which is relevant when there are nested loops or loops within loops).

This example, straight out of the GNU awk manual [2], shows a method of finding the smallest divisor. Read the additional comments for a clear understanding of how the code works:

```
#!/usr/bin/awk -f

{
    num = $1

    # Make an infinite FOR loop
    for (divisor = 2; ; divisor++) {

        # If num is divisible by divisor, then break
        if (num % divisor == 0) {
            printf "Smallest divisor of %d is %d\n",
                    num, divisor
            break
        }

        # If divisor has gotten too large, the number
        # has no divisor, so is a prime
        if (divisor * divisor > num) {
            printf "%d is prime\n", num
            break
        }
    }
}
```

Try running the script to see its results:

```
$ echo 67 | ./divisor.awk
67 is prime
$ echo 69 | ./divisor.awk
Smallest divisor of 69 is 3
```

As you can see, even though the script starts out with an explicit *infinite* loop with no end condition, the **break** function ensures that the script eventually terminates.

Continue

The **continue** function is similar to **break**. It can be used in a **for**, **while**, or **do-while** loop (it's not relevant to a **switch** statements, though). Invoking **continue** skips the rest of the enclosing loop and begins the next cycle.

Here's another good example from the GNU awk manual to demonstrate a possible use of **continue**:

```
#!/usr/bin/awk -f

# Loop, printing numbers 0-20, except 5

BEGIN {
    for (x = 0; x <= 20; x++) {
        if (x == 5)
            continue
        printf "%d ", x
    }
    print ""
}
```

This script analyzes the value of **x** before printing anything. If the value is exactly 5, then **continue** is invoked, causing the **printf** line to be skipped, but leaves the loop

unbroken. Try the same code but with **break** instead to see the difference.

Next

This statement is not related to loops like **break** and **continue** are. Instead, **next** applies to the main record processing cycle of awk: the functions you place between the BEGIN and END functions. The **next** statement causes awk to stop processing the *current input record* and to move to the next one.

As you know from the earlier articles in this series, awk reads records from its input stream and applies rules to them. The **next** statement stops the execution of rules for the current record and moves to the next one.

Here's an example of next being used to "hold" information upon a specific condition:

```
#!/usr/bin/awk -f

# Ignore the header
NR == 1 { next }

# If field 2 (colour) is less than 6
# characters, then save it with its
#  line number and skip it

length($2) < 6 {
    skip[NR] = $0
    next
}

# It's not the header and
# the colour name is > 6 characters,
# so print the line
{
    print
}

# At the end, show what was skipped
END {
    printf "\nSkipped:\n"
    for (n in skip)
        print n": "skip[n]
}
```

This sample uses next in the first rule to avoid the first line of the file, which is a header row. The second rule skips lines when the color name is less than six characters long, but it also saves that line in an array called skip, using the line number as the key (also known as the index).

The third rule prints anything it sees, but it is not invoked if either rule 1 or rule 2 causes it to be skipped.

Finally, at the end of all the processing, the END rule prints the contents of the array.

Run the sample script on the colours.txt file from above (and previous articles):

```
$ ./next.awk colours.txt
banana     yellow 6
grape      purple 10
plum       purple 2
pineapple  yellow 5

Skipped:
2: apple      red    4
4: strawberry red    3
6: apple      green  8
8: kiwi       brown  4
9: potato     brown  9
```

Control freak

In summary, **switch**, **continue**, **next**, and **break** are important preemptive exceptions to awk rules that provide greater control of your script. You don't have to use them directly; often, you can gain the same logic through other means, but they're great convenience functions that make the coder's life a lot easier. The next article in this series covers the **printf** statement.

Links

[1] https://opensource.com/article/19/11/loops-awk
[2] https://www.gnu.org/software/gawk/manual/

Advance your awk skills with two easy tutorials

Go beyond one-line awk scripts with mail merge and word counting.

AWK IS ONE OF THE OLDEST TOOLS in the Unix and Linux user's toolbox. Created in the 1970s by Alfred Aho, Peter Weinberger, and Brian Kernighan (the A, W, and K of the tool's name), awk was created for complex processing of text streams. It is a companion tool to sed, the stream editor, which is designed for line-by-line processing of text files. Awk allows more complex structured programs and is a complete programming language.

This article will explain how to use awk for more structured and complex tasks, including a simple mail merge application.

Awk program structure

An awk script is made up of functional blocks surrounded by **{}** (curly brackets). There are two special function blocks, **BEGIN** and **END**, that execute before processing the first line of the input stream and after the last line is processed. In between, blocks have the format:

```
pattern { action statements }
```

Each block executes when the line in the input buffer matches the pattern. If no pattern is included, the function block executes on every line of the input stream.

Also, the following syntax can be used to define functions in awk that can be called from any block:

```
function name(parameter list) { statements }
```

This combination of pattern-matching blocks and functions allows the developer to structure awk programs for reuse and readability.

How awk processes text streams

Awk reads text from its input file or stream one line at a time and uses a field separator to parse it into a number of fields. In awk terminology, the current buffer is a *record*. There are a number of special variables that affect how awk reads and processes a file:

- **FS** (field separator): By default, this is any whitespace (spaces or tabs)
- **RS** (record separator): By default, a newline (\n)
- **NF** (number of fields): When awk parses a line, this variable is set to the number of fields that have been parsed
- **$0**: The current record
- **$1**, **$2**, **$3**, etc.: The first, second, third, etc. field from the current record
- **NR** (number of records): The number of records that have been parsed so far by the awk script

There are many other variables that affect awk's behavior, but this is enough to start with.

Awk one-liners

For a tool so powerful, it's interesting that most of awk's usage is basic one-liners. Perhaps the most common awk program prints selected fields from an input line from a CSV file, a log file, etc. For example, the following one-liner prints a list of usernames from **/etc/passwd**:

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

As mentioned above, **$1** is the first field in the current record. The **-F** option sets the FS variable to the character :.

The field separator can also be set in a BEGIN function block:

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

In the following example, every user whose shell is not **/sbin/nologin** can be printed by preceding the block with a pattern match:

```
awk 'BEGIN { FS=":" } ! /\/sbin\/nologin/ {print $1 }'
     /etc/passwd
```

Advanced awk: Mail merge

Now that you have some of the basics, try delving deeper into awk with a more structured example: creating a mail merge.

A mail merge uses two files, one (called in this example **email_template.txt**) containing a template for an email you want to send:

```
From: Program committee <pc@event.org>
To: {firstname} {lastname} <{email}>
Subject: Your presentation proposal

Dear {firstname},

Thank you for your presentation proposal:
  {title}

We are pleased to inform you that your proposal has
been successful! We
will contact you shortly with further information about
the event
schedule.

Thank you,
The Program Committee
```

And the other is a CSV file (called **proposals.csv**) with the people you want to send the email to:

```
firstname,lastname,email,title
Harry,Potter,hpotter@hogwarts.edu,"Defeating your
  nemesis in 3 easy steps"
Jack,Reacher,reacher@covert.mil,"Hand-to-hand combat
  for beginners"
Mickey,Mouse,mmouse@disney.com,"Surviving public
  speaking with a squeaky voice"
Santa,Claus,sclaus@northpole.org,"Efficient list-making"
```

You want to read the CSV file, replace the relevant fields in the first file (skipping the first line), then write the result to a file called **acceptanceN.txt**, incrementing **N** for each line you parse.

Write the awk program in a file called **mail_merge.awk**. Statements are separated by **;** in awk scripts. The first task is to set the field separator variable and a couple of other variables the script needs. You also need to read and discard the first line in the CSV, or a file will be created starting with *Dear firstname*. To do this, use the special function **getline** and reset the record counter to 0 after reading it.

```
BEGIN {
  FS=",";
  template="email_template.txt";
  output="acceptance";
  getline;
  NR=0;
}
```

The main function is very straightforward: for each line processed, a variable is set for the various fields—**firstname**, **lastname**, **email**, and **title**. The template file is read line by line, and the function **sub** is used to substitute any occurrence of the special character sequences with the value of the relevant variable. Then the line, with any substitutions made, is output to the output file.

Since you are dealing with the template file and a different output file for each line, you need to clean up and close the file handles for these files before processing the next record.

```
{
        # Read relevant fields from input file
        firstname=$1;
        lastname=$2;
        email=$3;
        title=$4;

        # Set output filename
        outfile=(output NR ".txt");

        # Read a line from template, replace special
        # fields, and print result to output file
        while ( (getline ln < template) > 0 )
        {
                sub(/{firstname}/,firstname,ln);
                sub(/{lastname}/,lastname,ln);
                sub(/{email}/,email,ln);
                sub(/{title}/,title,ln);
                print(ln) > outfile;
        }

        # Close template and output file in advance of
next record
        close(outfile);
        close(template);
}
```

You're done! Run the script on the command line with:

```
awk -f mail_merge.awk proposals.csv
```

or

```
awk -f mail_merge.awk < proposals.csv
```

and you will find text files generated in the current directory.

Advanced awk: Word frequency count

One of the most powerful features in awk is the associative array. In most programming languages, array entries are typically indexed by a number, but in awk, arrays are referenced by a key string. You could store an entry from the file *proposals.txt* from the previous section. For example, in a single associative array, like this:

```
        proposer["firstname"]=$1;
        proposer["lastname"]=$2;
        proposer["email"]=$3;
        proposer["title"]=$4;
```

This makes text processing very easy. A simple program that uses this concept is the idea of a word frequency counter. You can parse a file, break out words (ignoring punctuation) in each line, increment the counter for each word in the line, then output the top 20 words that occur in the text.

First, in a file called **wordcount.awk**, set the field separator to a regular expression that includes whitespace and punctuation:

```
BEGIN {
        # ignore 1 or more consecutive occurrences of
the characters
        # in the character group below
        FS="[ .,:;()<>{}@!\"'\t]+";
}
```

Next, the main loop function will iterate over each field, ignoring any empty fields (which happens if there is punctuation at the end of a line), and increment the word count for the words in the line.

```
{
        for (i = 1; i <= NF; i++) {
                if ($i != "") {
                        words[$i]++;
                }
        }
}
```

Finally, after the text is processed, use the END function to print the contents of the array, then use awk's capability of piping output into a shell command to do a numerical sort and print the 20 most frequently occurring words:

```
END {
        sort_head = "sort -k2 -nr | head -n 20";
        for (word in words) {
                printf "%s\t%d\n", word, words[word] |
                  sort_head;
        }
        close (sort_head);
}
```

Running this script on an earlier draft of this article produced this output:

```
[dneary@dhcp-49-32.bos.redhat.com]$ awk -f wordcount.
awk < awk_article.txt
the     79
awk     41
a       39
and     33
of      32
in      27
to      26
is      25
line    23
for     23
will    22
file    21
we      16
We      15
with    12
which   12
by      12
this    11
output  11
function        11
```

What's next?

If you want to learn more about awk programming, I strongly recommend the book *Sed* and *awk* [1] by Dale Dougherty and Arnold Robbins.

One of the keys to progressing in awk programming is mastering "extended regular expressions." Awk offers several powerful additions to the sed regular expression [2] syntax you may already be familiar with.

Another great resource for learning awk is the GNU awk user guide [3]. It has a full reference for awk's built-in function library, as well as lots of examples of simple and complex awk scripts.

Links

[1] https://www.amazon.com/sed-awk-Dale-Dougherty/dp/1565922255/book

[2] https://en.wikibooks.org/wiki/Regular_Expressions/POSIX-Extended_Regular_Expressions

[3] https://www.gnu.org/software/gawk/manual/gawk.html

How to remove duplicate lines from files with awk

Learn how to use awk '!visited[$0]++' without sorting or changing their order.

SUPPOSE YOU HAVE a text file and you need to remove all of its duplicate lines.

TL;DR

To remove the duplicate lines while preserving their order in the file, use:

```
awk '!visited[$0]++' your_file > deduplicated_file
```

How it works

The script keeps an associative array with *indices* equal to the unique lines of the file and *values* equal to their occurrences. For each line of the file, if the line occurrences are zero, then it increases them by one and *prints the line*, otherwise, it just increases the occurrences *without printing the line*.

I was not familiar with **awk**, and I wanted to understand how this can be accomplished with such a short script (**awk**ward). I did my research, and here is what is going on:

- The awk "script" **!visited[$0]++** is executed for each *line* of the input file.
- **visited[]** is a variable of type associative array [1] (a.k.a. Map [2]). We don't have to initialize it because **awk** will do it the first time we access it.
- The **$0** variable holds the contents of the line currently being processed.
- **visited[$0]** accesses the value stored in the map with a key equal to $0 (the line being processed), a.k.a. the occurrences (which we set below).
- The **!** negates the occurrences' value:
  - In awk, any nonzero numeric value or any nonempty string value is true [3].
  - By default, variables are initialized to the empty string [4], which is zero if converted to a number.
  - That being said:
    - If **visited[$0]** returns a number greater than zero, this negation is resolved to **false**.
    - If **visited[$0]** returns a number equal to zero or an empty string, this negation is resolved to **true**.
- The **++** operation increases the variable's value (**visited[$0]**) by one.
  - If the value is empty, **awk** converts it to **0** (number) automatically and then it gets increased.
  - **Note:** The operation is executed after we access the variable's value.

Summing up, the whole expression evaluates to:

- **true** if the occurrences are zero/empty string
- **false** if the occurrences are greater than zero

awk statements consist of a *pattern-expression* and an *associated action* [5].

```
<pattern/expression> { <action> }
```

If the pattern succeeds, then the associated action is executed. If we don't provide an action, **awk**, by default, **prints** the input.

> An omitted action is equivalent to **{ print $0 }**.

Our script consists of one **awk** statement with an expression, omitting the action. So this:

```
awk '!visited[$0]++' your_file > deduplicated_file
```

is equivalent to this:

```
awk '!visited[$0]++ { print $0 }' your_file >
  deduplicated_file
```

For every line of the file, if the expression succeeds, the line is printed to the output. Otherwise, the action is not executed, and nothing is printed.

Why not use the uniq command?

The **uniq** command removes only the *adjacent duplicate lines*. Here's a demonstration:

```
$ cat test.txt
A
A
A
B
B
B
A
A
C
C
C
B
B
A
$ uniq < test.txt
A
B
A
C
B
A
```

Other approaches

Using the sort command

We can also use the following **sort** [6] command to remove the duplicate lines, but *the line order is not preserved.*

```
sort -u your_file > sorted_deduplicated_file
```

Using cat, sort, and cut

The previous approach would produce a de-duplicated file whose lines would be sorted based on the contents. Piping a bunch of commands [7] can overcome this issue:

```
cat -n your_file | sort -uk2 | sort -nk1 | cut -f2-
```

How it works

Suppose we have the following file:

```
abc
ghi
abc
def
xyz
def
ghi
klm
```

cat -n test.txt prepends the order number in each line.

```
1       abc
2       ghi
3       abc
4       def
5       xyz
6       def
7       ghi
8       klm
```

sort -uk2 sorts the lines based on the second column (**k2** option) and keeps only the first occurrence of the lines with the same second column value (**u** option).

```
1       abc
4       def
2       ghi
8       klm
5       xyz
```

sort -nk1 sorts the lines based on their first column (**k1** option) treating the column as a number (**-n** option).

```
1       abc
2       ghi
4       def
5       xyz
8       klm
```

Finally, **cut -f2-** prints each line starting from the second column until its end (**-f2-** option: *Note the - suffix, which instructs it to include the rest of the line*).

```
abc
ghi
def
xyz
klm
```

References

- The GNU awk user's guide
- Arrays in awk
- Awk—Truth values
- Awk expressions
- How can I delete duplicate lines in a file in Unix?
- Remove duplicate lines without sorting [duplicate]
- How does awk '!a[$0]++' work?

Links

[1] http://kirste.userpage.fu-berlin.de/chemnet/use/info/gawk/gawk_12.html
[2] https://en.wikipedia.org/wiki/Associative_array
[3] https://www.gnu.org/software/gawk/manual/html_node/Truth-Values.html
[4] https://ftp.gnu.org/old-gnu/Manuals/gawk-3.0.3/html_chapter/gawk_8.html
[5] http://kirste.userpage.fu-berlin.de/chemnet/use/info/gawk/gawk_9.html
[6] http://man7.org/linux/man-pages/man1/sort.1.html
[7] https://stackoverflow.com/a/20639730/2292448

Awk one-liners and scripts to help you sort text files

Awk is a powerful tool for doing tasks that might otherwise be left to other common utilities, including sort.

AWK IS THE UBIQUITOUS UNIX COMMAND for scanning and processing text containing predictable patterns. However, because it features functions, it's also justifiably called a programming language..

Confusingly, there is more than one awk. (Or, if you believe there can be only one, then there are several clones.) There's **awk**, the original program written by Aho, Weinberger, and Kernighan, and then there's **nawk**, **mawk**, and the GNU version, **gawk**. The GNU version of awk is a highly portable, free software version of the utility with several unique features, so this article is about GNU awk.

While its official name is gawk, on GNU+Linux systems it's aliased to awk and serves as the default version of that command. On other systems that don't ship with GNU awk, you must install it and refer to it as gawk, rather than awk. This article uses the terms awk and gawk interchangeably.

Being both a command and a programming language makes awk a powerful tool for tasks that might otherwise be left to **sort**, **cut**, **uniq**, and other common utilities. Luckily, there's lots of room in open source for redundancy, so if you're faced with the question of whether or not to use awk, the answer is probably a solid "maybe."

The beauty of awk's flexibility is that if you've already committed to using awk for a task, then you can probably stay in awk no matter what comes up along the way. This includes the eternal need to sort data in a way other than the order it was delivered to you.

Sample set

Before exploring awk's sorting methods, generate a sample dataset to use. Keep it simple so that you don't get distracted by edge cases and unintended complexity. This is the sample set this article uses:

```
Aptenodytes;forsteri;Miller,JF;1778;Emperor
Pygoscelis;papua;Wagler;1832;Gentoo
Eudyptula;minor;Bonaparte;1867;Little Blue
Spheniscus;demersus;Brisson;1760;African
Megadyptes;antipodes;Milne-Edwards;1880;Yellow-eyed
Eudyptes;chrysocome;Viellot;1816;Sothern Rockhopper
Torvaldis;linux;Ewing,L;1996;Tux
```

It's a small dataset, but it offers a good variety of data types:

- A genus and species name, which are associated with one another but considered separate
- A surname, sometimes with first initials after a comma
- An integer representing a date
- An arbitrary term
- All fields separated by semi-colons

Depending on your educational background, you may consider this a 2D array or a table or just a line-delimited collection of data. How you think of it is up to you, because awk doesn't expect anything more than text. It's up to you to tell awk how you want to parse it.

The sort cheat

If you just want to sort a text dataset by a specific, definable field (think of a "cell" in a spreadsheet), then you can use the sort command [1].

Fields and records

Regardless of the format of your input, you must find patterns in it so that you can focus on the parts of the data that are important to you. In this example, the data is delimited by two factors: lines and fields. Each new line represents a new record, as you would likely see in a spreadsheet or database dump. Within each line, there are distinct *fields* (think of them as cells in a spreadsheet) that are separated by semicolons (;).

Awk processes one record at a time, so while you're structuring the instructions you will give to awk, you can focus on just

one line. Establish what you want to do with one line, then test it (either mentally or with awk) on the next line and a few more. You'll end up with a good hypothesis on what your awk script must do in order to provide you with the data structure you want.

In this case, it's easy to see that each field is separated by a semicolon. For simplicity's sake, assume you want to sort the list by the very first field of each line.

Before you can sort, you must be able to focus awk on just the first field of each line, so that's the first step. The syntax of an awk command in a terminal is **awk**, followed by relevant options, followed by your awk command, and ending with the file of data you want to process.

```
$ awk --field-separator=";" '{print $1;}' penguins.list
Aptenodytes
Pygoscelis
Eudyptula
Spheniscus
Megadyptes
Eudyptes
Torvaldis
```

Because the field separator is a character that has special meaning to the Bash shell, you must enclose the semicolon in quotes or precede it with a backslash. This command is useful only to prove that you can focus on a specific field. You can try the same command using the number of another field to view the contents of another "column" of your data:

```
$ awk --field-separator=";" '{print $3;}' penguins.list
Miller,JF
Wagler
Bonaparte
Brisson
Milne-Edwards
Viellot
Ewing,L
```

Nothing has been sorted yet, but this is good groundwork.

Scripting

Awk is more than just a command; it's a programming language with indices and arrays and functions. That's significant because it means you can grab a list of fields you want to sort by, store the list in memory, process it, and then print the resulting data. For a complex series of actions such as this, it's easier to work in a text file, so create a new file called **sorter.awk** and enter this text:

```
#!/usr/bin/awk -f

BEGIN {
        FS=";";
}
```

This establishes the file as an awk script that executes the lines contained in the file.

The **BEGIN** statement is a special setup function provided by awk for tasks that need to occur only once. Defining the built-in variable **FS**, which stands for field separator and is the same value you set in your awk command with **--field-separator**, only needs to happen once, so it's included in the **BEGIN** statement.

Arrays in awk

You already know how to gather the values of a specific field by using the **$** notation along with the field number, but in this case, you need to store it in an array rather than print it to the terminal. This is done with an awk array. The important thing about an awk array is that it contains keys and values. Imagine an array about this article; it would look something like this: **author:"seth",title:"How to sort with awk",length:1200**. Elements like **author** and **title** and **length** are keys, with the following contents being values.

The advantage to this in the context of sorting is that you can assign any field as the key and any record as the value, and then use the built-in awk function **asorti()** (sort by index) to sort by the key. For now, assume arbitrarily that you *only* want to sort by the second field.

Awk statements not preceded by the special keywords **BEGIN** or **END** are loops that happen at each record. This is the part of the script that scans the data for patterns and processes it accordingly. Each time awk turns its attention to a record, statements in **{}** (unless preceded by **BEGIN** or **END**) are executed.

To add a key and value to an array, create a variable (in this example script, I call it **ARRAY**, which isn't terribly original, but very clear) containing an array, and then assign it a key in brackets and a value with an equals sign (**=**).

```
{   # dump each field into an array
    ARRAY[$2] = $R;
}
```

In this statement, the contents of the second field (**$2**) are used as the key term, and the current record (**$R**) is used as the value.

The asorti() function

In addition to arrays, awk has several basic functions that you can use as quick and easy solutions for common tasks. One of the functions introduced in GNU awk, **asorti()**, provides the ability to sort an array by key (or *index*) or value.

You can only sort the array once it has been populated, meaning that this action must not occur with every new record but only the final stage of your script. For this purpose, awk provides the special **END** keyword. The inverse of **BEGIN**, an **END** statement happens only once and only after all records have been scanned.

Add this to your script:

```
END {
    asorti(ARRAY,SARRAY);
    # get length
    j = length(SARRAY);

    for (i = 1; i <= j; i++) {
        printf("%s %s\n", SARRAY[i],ARRAY[SARRAY[i]])
    }
}
```

The **asorti()** function takes the contents of **ARRAY**, sorts it by index, and places the results in a new array called **SARRAY** (an arbitrary name I invented for this article, meaning *Sorted ARRAY*).

Next, the variable **j** (another arbitrary name) is assigned the results of the **length()** function, which counts the number of items in **SARRAY**.

Finally, use a **for** loop to iterate through each item in **SARRAY** using the **printf()** function to print each key, followed by the corresponding value of that key in **ARRAY**.

Running the script

To run your awk script, make it executable:

```
$ chmod +x sorter.awk
```

And then run it against the **penguin.list** sample data:

```
$ ./sorter.awk penguins.list
antipodes Megadyptes;antipodes;Milne-Edwards;1880;
  Yellow-eyed
chrysocome Eudyptes;chrysocome;Viellot;1816;
  Sothern Rockhopper
demersus Spheniscus;demersus;Brisson;1760;African
forsteri Aptenodytes;forsteri;Miller,JF;1778;Emperor
linux Torvaldis;linux;Ewing,L;1996;Tux
minor Eudyptula;minor;Bonaparte;1867;Little Blue
papua Pygoscelis;papua;Wagler;1832;Gentoo
```

As you can see, the data is sorted by the second field.

This is a little restrictive. It would be better to have the flexibility to choose at runtime which field you want to use as your sorting key so you could use this script on any dataset and get meaningful results.

Adding command options

You can add a command variable to an awk script by using the literal value **var** in your script. Change your script so that your iterative clause uses **var** when creating your array:

```
{ # dump each field into an array
    ARRAY[$var] = $R;
}
```

Try running the script so that it sorts by the third field by using the **-v var** option when you execute it:

```
$ ./sorter.awk -v var=3 penguins.list
Bonaparte Eudyptula;minor;Bonaparte;1867;Little Blue
Brisson Spheniscus;demersus;Brisson;1760;African
Ewing,L Torvaldis;linux;Ewing,L;1996;Tux
Miller,JF Aptenodytes;forsteri;Miller,JF;1778;Emperor
Milne-Edwards Megadyptes;antipodes;Milne-Edwards;1880;
  Yellow-eyed
Viellot Eudyptes;chrysocome;Viellot;1816;
  Sothern Rockhopper
Wagler Pygoscelis;papua;Wagler;1832;Gentoo
```

Fixes

This article has demonstrated how to sort data in pure GNU awk. The script can be improved so, if it's useful to you, spend some time researching awk functions [2] on gawk's man page and customizing the script for better output.

Here is the complete script so far:

```
#!/usr/bin/awk -f
# GPLv3 appears here
# usage: ./sorter.awk -v var=NUM FILE

BEGIN { FS=";"; }

{ # dump each field into an array
    ARRAY[$var] = $R;
}

END {
    asorti(ARRAY,SARRAY);
    # get length
    j = length(SARRAY);

    for (i = 1; i <= j; i++) {
        printf("%s %s\n", SARRAY[i],ARRAY[SARRAY[i]])
    }
}
```

Links

[1] https://opensource.com/article/19/10/get-sorted-sort
[2] https://www.gnu.org/software/gawk/manual/html_node/Built_002din.html#Built_002din

A gawk script to convert smart quotes

I MANAGE a personal website and edit the web pages by hand. Since I don't have many pages on my site, this works well for me, letting me "scratch the itch" of getting into the site's code.

When I updated my website's design recently, I decided to turn all the plain quotes into "smart quotes," or quotes that look like those used in print material: “” instead of "".

Editing all of the quotes by hand would take too long, so I decided to automate the process of converting the quotes in all of my HTML files. But doing so via a script or program requires some intelligence. The script needs to know when to convert a plain quote to a smart quote, and which quote to use.

You can use different methods to convert quotes. Greg Pittman wrote a Python script [1] for fixing smart quotes in text. I wrote mine in GNU awk (gawk) [2].

To start, I wrote a simple gawk function to evaluate a single character. If that character is a quote, the function determines if it should output a plain quote or a smart quote. The function looks at the previous character; if the previous character is a space, the function outputs a left smart quote. Otherwise, the function outputs a right smart quote. The script does the same for single quotes.

```
function smartquote (char, prevchar) {
        # print smart quotes depending on the previous
        # character otherwise just print the character as-is

        if (prevchar ~ /\s/) {
                # prev char is a space
                if (char == "'") {
                        printf("&lsquo;");
                }
                else if (char == "\"") {
                        printf("&ldquo;");
                }
                else {
                        printf("%c", char);
                }
        }
        else {
                # prev char is not a space
                if (char == "'") {
                        printf("&rsquo;");
                }
                else if (char == "\"") {
                        printf("&rdquo;");
                }
                else {
                        printf("%c", char);
                }
        }
}
```

With that function, the body of the gawk script processes the HTML input file character by character. The script prints all text verbatim when inside an HTML tag (for example, `<html lang="en">`. Outside any HTML tags, the script uses the `smartquote()` function to print text. The smartquote() function does the work of evaluating when to print plain quotes or smart quotes.

```
function smartquote (char, prevchar) {
        ...
}

BEGIN {htmltag = 0}

{
        # for each line, scan one letter at a time:

        linelen = length($0);

        prev = "\n";

        for (i = 1; i <= linelen; i++) {
                char = substr($0, i, 1);
```

```
                if (char == "<") {
                        htmltag = 1;
                }

                if (htmltag == 1) {
                        printf("%c", char);
                }
                else {
                        smartquote(char, prev);
                        prev = char;
                }

                if (char == ">") {
                        htmltag = 0;
                }
        }

        # add trailing newline at end of each line
        printf ("\n");
}
```

Here's an example:

```
gawk -f quotes.awk test.html > test2.html
Sample input:

<!DOCTYPE html>
<html lang="en">
<head>
  <title>Test page</title>
  <link rel="stylesheet" type="text/css" href="/test.css" />
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width" />
</head>
<body>
  <h1><a href="/"><img src="logo.png" alt="Website logo" />
      </a></h1>
  <p>"Hi there!"</p>
  <p>It's and its.</p>
</body>
</html>
Sample output:

<!DOCTYPE html>
<html lang="en">
<head>
  <title>Test page</title>
  <link rel="stylesheet" type="text/css" href="/test.css" />
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width" />
</head>
<body>
  <h1><a href="/"><img src="logo.png" alt="Website logo" />
      </a></h1>
  <p>&ldquo;Hi there!&rdquo;</p>
  <p>It&rsquo;s and its.</p>
</body>
</html>
```

Links

[1] https://opensource.com/article/17/3/python-scribus-smart-quotes

[2] https://opensource.com/downloads/cheat-sheet-awk-features

Drinking coffee with AWK

Keep track of what your office mates owe for the coffee they drink with a simple AWK program.

THE FOLLOWING is based on a true story, although some names and details have been changed.

> A long time ago, in a place far away, there was an office. The office did not, for various reasons, buy instant coffee. Some workers in that office got together and decided to institute the "Coffee Corner."
>
> A member of the Coffee Corner would buy some instant coffee, and the other members would pay them back. It came to pass that some people drank more coffee than others, so the level of a "half-member" was added: a half-member was allowed a limited number of coffees per week and would pay half of what a member paid

Managing this was a huge pain. I had just read *The Unix Programming Environment* and wanted to practice my AWK [1] programming. So I volunteered to create a system.

Step 1: I kept a database of members and their debt to the Coffee Corner. I did it in an AWK-friendly format, where fields are separated by colons:

```
member:john:1:22
member:jane:0.5:33
member:pratyush:0.5:17
member:jing:1:27
```

The first field above identifies what kind of row this is (member). The second field is the member's name (i.e., their email username without the @). The next field is their membership level (full=1 or half=0.5). The last field is their debt to the Coffee Corner. A positive number means they owe money, a negative number means the Coffee Corner owes them.

Step 2: I kept a log of inputs to and outputs from the Coffee Corner:

```
payment:jane:33
payment:pratyush:17
bought:john:60
payback:john:50
```

Jane paid $33, Pratyush paid $17, John bought $60 worth of coffee, and the Coffee Corner paid John $50.

Step 3: I was ready to write some code. The code would process the members and payments and spit out an updated members file with the new debts.

```
#!/usr/bin/env --split-string=awk -F: -f
```

The shebang (**#!**) line required some work! I used the **env** command to allow passing multiple arguments from the shebang: specifically, the **-F** command-line argument to AWK tells it what the field separator is.

An AWK program is a sequence of rules. (It can also contain function definitions, but I don't need any for the Coffee Corner.)

The first rule reads the *members* file. When I run the command, I always give it the *members* file first, and the *payments* file second. It uses AWK associative arrays to record membership levels in the **members** array and current **debt** in the debt array.

```
$1 == "member" {
   members[$2]=$3
   debt[$2]=$4
   total_members += $3
}
```

The second rule reduces the debt when a **payment** is recorded.

```
$1 == "payment" {
   debt[$2] -= $3
}
```

Payback is the opposite: it *increases* the debt. This elegantly supports the case of accidentally giving someone too much money.

```
$1 == "payback" {
   debt[$2] += $3
}
```

The most complicated part happens when someone buys (**"bought"**) instant coffee for the Coffee Club's use. It is treated as a payment and the person's debt is reduced by the appropriate amount. Next, it calculates the per-member fee. It iterates over all members and increases their debt, according to their level of membership.

```
$1 == "bought" {
   debt[$2] -= $3
   per_member = $3/total_members
   for (x in members) {
       debt[x] += per_member * members[x]
   }
}
```

The **END** pattern is special: it happens exactly once, when AWK has no more lines to process. At this point, it spits out the new *members* file with updated debt levels.

```
END {
   for (x in members) {
       printf "%s:%s:%s\n", x, members[x], debt[x]
   }
}
```

Along with a script that iterates over the members and sends a reminder email to people to pay their dues (for positive debts), this system managed the Coffee Corner for quite a while.

Links

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

Opensource.com: GNU awk Cheat Sheet BY JIM HALL

Use this handy quick reference guide to the most commonly used features of GNU awk (gawk).

COMMAND-LINE USAGE

Run a gawk script using **-f** or include a short script right on the command line.

gawk -f file.awk file1 file2…

or:

gawk 'pattern {action}' file1 file2…

also: set the field separator using **-F**

gawk -F: …

PATTERNS

All program lines are some combination of a pattern and actions:

pattern {action}

where **pattern** can be:

- **BEGIN** (matches start of input)
- **END** (matches end of input)
- a regular expression (act only on matching lines)
- a comparison (act only when true)
- empty (act on all lines)

ACTIONS

Actions are very similar to C programming.

Actions can span multiple lines.

End statements with a semicolon (**;**)

For example:

```
BEGIN { FS = ":"; }

{ print "Hello world"; }

{
  print;
  i = i + 1;
}
```

FIELDS

Gawk does the work for you and splits input lines so you can reference them by field. Use **-F** on the command line or set **FS** to set the field separator.

- Reference fields using **$**
- **$1** for the first string, and so on
- Use **$0** for the entire line

For example:

gawk '{print "1st word:", $1;}' file.txt

or:

gawk -F: '{print "uid", $3;}' /etc/passwd

REGULAR EXPRESSIONS

Common regular expression patterns include:

^	Matches start of a line
$	Matches end of a line
.	Matches any character, including newline
a	Matches a single letter **a**
a+	Matches one or more **a**'s
a*	Matches zero or more **a**'s
a?	Matches zero or one **a**'s
[abc]	Matches any of the characters **a**, **b**, or **c**
[^abc]	Negation; matches any character except **a**, **b**, or **c**
\.	Use backslash (\) to match a special character (like **.**)

You can also use character classes, including:

[:alpha:]	Any alphabetic character
[:lower:]	Any lowercase letter
[:upper:]	Any uppercase letter
[:digit:]	Any numeric character
[:alnum:]	Any alphanumeric character
[:cntrl:]	Any control character
[:blank:]	Spaces or tabs
[:space:]	Spaces, tabs, and other white space (such as linefeed)

OPERATORS

(…)	Grouping
++ --	Increment and decrement
^	Exponents
+ - !	Unary plus, minus, and negation
*** / %**	Multiply, divide, and modulo
+ -	Add and subtract
< > <= >= == !=	Relations
~ !~	Regular expression match or negated match
&&	Logical AND
ll	Logical OR
= += -= *= /= %= ^=	Assignment

FLOW CONTROL

You can use many common flow control and loop structures, including if, while, do-while, for, and switch.

```
if (i < 10) { print; }

while (i < 10) { print; i++; }

do {
  print;
  i++;
} while (i < 10);

for (i = 1; i < 10; i++) { print i; }

switch (n) {
  case 1: print "yes";
  ⋮
  default: print "no";
}
```

FUNCTIONS

Frequently-used string functions include:

```
print "hello world"
print "user:" $1
print $1, $2
print i
print
```

Print a value or string. If you don't give a value, outputs **$0** instead.

Use commas (,) to put space between the values.

Use spaces ( ) to combine the output.

printf(fmt, values…)

The standard C **printf** function.

sprintf(fmt, values…)

Similar to the standard C **sprintf** function, returns the new string.

index(str, sub)

Return the index of the substring **sub** in the string **str**, or zero if not found.

length([str])

Return the length of the string **$0**.

If you include the string **str**, give that length instead.

FUNCTIONS (CONTINUED)

substr(str, pos [, n])

Return the next **n** characters of the string **str**, starting at position **pos**.

If **n** is omitted, return the rest of the string **str**.

tolower(str)

Return the string **str**, converted to all lowercase.

toupper(str)

Return the string **str**, converted to all uppercase.

Other common string functions include:

match(str, regex)

Return the position of the first occurrence of the regular expression **regex** in the string **str**.

sub(sub, repl [, str])

For the first matching substring **sub** in the string **$0**, replace it with **repl**.

If you include the optional string **str**, operate on that string instead.

gsub(sub, repl [, str])

Same as **sub()**, but replaces all matching substrings.

split(str, arr [, del])

Splits up the string **str** into the array **arr**, according to spaces and tabs.

If you include the optional string **del**, use that as the field delimiter characters.

strtonum(str)

Return the numeric value of the string **str**. Works with decimal, octal, and hexadecimal values.

USER-DEFINED FUNCTIONS

You can define your own functions to add new functionality, or to make frequently-used code easier to reference.

Define a function using the **function** keyword:

```
function name(parameters) {
  statements
}
```

