GNU AWK

{awesome one-liners}

```
'BEGIN{n=100} {n += $NF} END{print n}'
'{gsub(/foo/, "bar", $1)} 1'
'{print $4, $2+7}'
'!a[$0]++'
-F'[0-9]+' 'NF==4'
-v RS= -v ORS='\n\n' '/baz/'
'NR==FNR{a[$1,$2]; next} ($1,$2) in a'
```

Sundeep Agarwal

Table of contents

Preface

When it comes to command line text processing, from an abstract point of view, there are three major pillars — `grep` for filtering, `sed` for substitution and `awk` for field processing. These tools have some overlapping features too, for example, all three of them have extensive filtering capabilities.

Unlike `grep` and `sed` , `awk` is a full blown programming language. However, this book intends to showcase `awk` one-liners that can be composed from the command line instead of writing a program file.

This book heavily leans on examples to present options and features of `awk` one by one. It is recommended that you manually type each example and experiment with them. Understanding both the nature of sample input string and the output produced is essential. As an analogy, consider learning to drive a bike or a car — no matter how much you read about them or listen to explanations, you need to practice a lot and infer your own conclusions. Should you feel that copy-paste is ideal for you, code snippets are available chapter wise on GitHub.

My Command Line Text Processing repository includes a chapter on `GNU awk` which has been edited and restructured to create this book.

Prerequisites

- Prior experience working with command line and `bash` shell, should know concepts like file redirection, command pipeline and so on
- Familiarity with programming concepts like variables, printing, functions, control structures, arrays, etc
- Knowing basics of `grep` and `sed` will help in understanding similar features of `awk`

If you are new to the world of command line, check out ryanstutorials or my GitHub repository on Linux Command Line before starting this book.

Conventions

- The examples presented here have been tested on `GNU bash` shell with **GNU awk 5.0.1** and includes features not available in earlier versions
- Code snippets shown are copy pasted from `bash` shell and modified for presentation purposes. Some commands are preceded by comments to provide context and explanations. Blank lines have been added to improve readability, only `real` time is shown for speed comparisons, output is skipped for commands like `wget` and so on
- Unless otherwise noted, all examples and explanations are meant for **ASCII** characters
- `awk` would mean `GNU awk` , `grep` would mean `GNU grep` and so on unless otherwise specified
- External links are provided for further reading throughout the book. Not necessary to immediately visit them. They have been chosen with care and would help, especially during re-reads
- The learn_gnuawk repo has all the files used in examples and exercises and other details related to the book. If you are not familiar with `git` command, click the **Clone or download** button on the webpage to get the files

Acknowledgements

- GNU awk documentation — manual and examples
- stackoverflow and unix.stackexchange — for getting answers to pertinent questions on `bash` , `awk` and other commands
- tex.stackexchange — for help on `pandoc` and `tex` related questions
- Cover image: LibreOffice Draw
- softwareengineering.stackexchange and skolakoda for programming quotes
- Warning and Info icons by Amada44 under public domain

Special thanks to all my friends and online acquaintances for their help, support and encouragement, especially during these difficult times.

Feedback and Errata

I would highly appreciate if you'd let me know how you felt about this book, it would help to improve this book as well as my future attempts. Also, please do let me know if you spot any error or typo.

Issue Manager: https://github.com/learnbyexample/learn_gnuawk/issues

Goodreads: https://www.goodreads.com/book/show/52758608-gnu-awk

E-mail: learnbyexample.net@gmail.com

Twitter: https://twitter.com/learn_byexample

Author info

Sundeep Agarwal is a freelance trainer, author and mentor. His previous experience includes working as a Design Engineer at Analog Devices for more than 5 years. You can find his other works, primarily focused on Linux command line, text processing, scripting languages and curated lists, at https://github.com/learnbyexample. He has also been a technical reviewer for Command Line Fundamentals book and video course published by Packt.

List of books: https://learnbyexample.github.io/books/

License

Book version

0.7
See Version_changes.md to track changes across book versions.

Installation and Documentation

The command name `awk` is derived from its developers — Alfred V. **A**ho, Peter J. **W**einberger, and Brian W. **K**ernighan. Over the years, it has been adapted and modified by various other developers. See gawk manual: History for more details. This chapter will show how to install or upgrade `awk` followed by details related to documentation.

Installation

If you are on a Unix like system, you are most likely to already have some version of `awk` installed. This book is primarily for `GNU awk` . As there are syntax and feature differences between various implementations, please make sure to follow along with what is presented here. `GNU awk` is part of text creation and manipulation commands provided by `GNU` . To install newer or particular version, visit gnu: software gawk. Check release notes for an overview of changes between versions.

```
$ # use a dir, say ~/Downloads/awk_install before following the steps below
$ wget https://ftp.gnu.org/gnu/gawk/gawk-5.0.1.tar.xz
$ tar -Jxf gawk-5.0.1.tar.xz
$ cd gawk-5.0.1/
$ ./configure
$ make
$ sudo make install

$ type -a awk
awk is /usr/local/bin/awk
awk is /usr/bin/awk
$ awk --version | head -n1
GNU Awk 5.0.1, API: 2.0
```

See also gawk manual: Installation for advanced options and instructions to install `awk` on other platforms.

Documentation

It is always a good idea to know where to find the documentation. From command line, you can use `man awk` for a short manual and `info awk` for full documentation. The online gnu awk manual has a better reading interface and provides the most complete documentation, examples and information about other `awk` versions, POSIX standard, etc.

Here's a snippet from `man awk` :

```
$ man awk
GAWK(1)                         Utility Commands                         GAWK(1)

NAME
       gawk - pattern scanning and processing language

SYNOPSIS
```

```
       gawk [ POSIX or GNU style options ] -f program-file [ -- ] file ...
       gawk [ POSIX or GNU style options ] [ -- ] program-text file ...

DESCRIPTION
       Gawk  is  the  GNU Project's implementation of the AWK programming lan-
       guage.  It conforms to the definition of  the  language  in  the  POSIX
       1003.1  Standard.   This version in turn is based on the description in
       The AWK Programming Language, by Aho, Kernighan, and Weinberger.   Gawk
       provides  the additional features found in the current version of Brian
       Kernighan's awk and a number of GNU-specific extensions.
```

Options overview

For a quick overview of all the available options, use `awk --help` from the command line.

```
$ # only partial output shown here and whitespace is adjusted for alignment
$ awk --help
Usage: awk [POSIX or GNU style options] -f progfile [--] file ...
Usage: awk [POSIX or GNU style options] [--] 'program' file ...
POSIX options:                  GNU long options: (standard)
    -f progfile                 --file=progfile
    -F fs                       --field-separator=fs
    -v var=val                  --assign=var=val
Short options:                  GNU long options: (extensions)
    -b                          --characters-as-bytes
    -c                          --traditional
    -C                          --copyright
    -d[file]                    --dump-variables[=file]
    -D[file]                    --debug[=file]
    -e 'program-text'           --source='program-text'
    -E file                     --exec=file
    -g                          --gen-pot
    -h                          --help
    -i includefile              --include=includefile
    -l library                  --load=library
    -L[fatal|invalid|no-ext]    --lint[=fatal|invalid|no-ext]
    -M                          --bignum
    -N                          --use-lc-numeric
    -n                          --non-decimal-data
    -o[file]                    --pretty-print[=file]
    -O                          --optimize
    -p[file]                    --profile[=file]
    -P                          --posix
    -r                          --re-interval
    -s                          --no-optimize
    -S                          --sandbox
    -t                          --lint-old
    -V                          --version
```

awk introduction

This chapter will give an overview of `awk` syntax and some examples to show what kind of problems you could solve using `awk` . These features will be covered in depth in later chapters, but don't go skipping this chapter.

Filtering

`awk` provides filtering capabilities like those supported by `grep` and `sed` plus some nifty features of its own. And similar to many command line utilities, `awk` can accept input from both `stdin` and files.

```
$ # sample stdin data
$ printf 'gate\napple\nwhat\nkite\n'
gate
apple
what
kite

$ # same as: grep 'at' and sed -n '/at/p'
$ # print all lines containing 'at'
$ printf 'gate\napple\nwhat\nkite\n' | awk '/at/'
gate
what

$ # same as: grep -v 'e' and sed -n '/e/!p'
$ # print all lines NOT containing 'e'
$ printf 'gate\napple\nwhat\nkite\n' | awk '!/e/'
what
```

Similar to `grep` and `sed` , by default `awk` automatically loops over input content line by line. You can then use `awk` 's programming instructions to process those lines. As `awk` is primarily used from the command line, many shortcuts are available to reduce the amount of typing needed.

In the above examples, a regular expression (defined by the pattern between a pair of forward slashes) has been used to filter the input. Regular expressions (regexp) will be covered in detail in next chapter, only simple string value is used here without any special characters. The full syntax is `string ~ /regexp/` to check if the given string matches the regexp and `string !~ /regexp/` to check if doesn't match. When the string isn't specified, the test is performed against a special variable `$0` , which has the contents of the input line. The correct term would be input **record**, but that's a discussion for a later chapter.

Also, in the above examples, only the filtering condition was given and nothing about what should be done. By default, when the condition evaluates to `true` , the contents of `$0` is printed. Thus:

- `awk '/regexp/'` is a shortcut for `awk '$0 ~ /regexp/{print $0}'`
- `awk '!/regexp/'` is a shortcut for `awk '$0 !~ /regexp/{print $0}'`

```
$ # same as: awk '/at/'
$ printf 'gate\napple\nwhat\nkite\n' | awk '$0 ~ /at/{print $0}'
gate
what

$ # same as: awk '!/e/'
$ printf 'gate\napple\nwhat\nkite\n' | awk '$0 !~ /e/{print $0}'
what
```

In the above examples, `{}` is used to specify a block of code to be executed when the condition that precedes the block evaluates to `true`. One or more statements can be given separated by `;` character. You'll see such examples and learn more about `awk` syntax later.

Any non-zero numeric value and non-empty string value is considered as `true` when that value is used as a conditional expression. Idiomatically, `1` is used to denote a `true` condition in one-liners as a shortcut to print the contents of `$0`.

```
$ # same as: printf 'gate\napple\nwhat\nkite\n' | cat
$ # same as: awk '{print $0}'
$ printf 'gate\napple\nwhat\nkite\n' | awk '1'
gate
apple
what
kite
```

Substitution

`awk` has three functions to cover search and replace requirements. Two of them are shown below. The `sub` function replaces only the first match whereas `gsub` function replaces all the matching occurrences. By default, these functions operate on `$0` when the input string isn't provided. Both `sub` and `gsub` modifies the input source on successful substitution.

```
$ # for each input line, change only first ':' to '-'
$ # same as: sed 's/:/-/'
$ printf '1:2:3:4\na:b:c:d\n' | awk '{sub(/:/, "-")} 1'
1-2:3:4
a-b:c:d

$ # for each input line, change all ':' to '-'
$ # same as: sed 's/:/-/g'
$ printf '1:2:3:4\na:b:c:d\n' | awk '{gsub(/:/, "-")} 1'
1-2-3-4
a-b-c-d
```

The first argument to `sub` and `gsub` functions is the regexp to be matched against the input content. The second argument is the replacement string. String literals are specified within double quotes. In the above examples, `sub` and `gsub` are used inside a block as they aren't intended to be used as a conditional expression. The `1` after the block is treated as a conditional expression as it is used outside a block. You can also use the variations presented below to get the same results.

- `awk '{sub(/:/, "-")} 1'` is same as `awk '{sub(/:/, "-"); print $0}'`
- You can also just use `print` instead of `print $0` as `$0` is the default string


You might wonder why to use or learn `grep` and `sed` when you can achieve same results with `awk`. It depends on the problem you are trying to solve. A simple line filtering will be faster with `grep` compared to `sed` or `awk` because `grep` is optimized for such cases. Similarly, `sed` will be faster than `awk` for substitution cases. Also, not all features easily translate among these tools. For example, `grep -o` requires lot more steps to code with `sed` or `awk`. Only `grep` offers recursive search. And so on. See also unix.stackexchange: When to use grep, sed, awk, perl, etc.

Field processing

As mentioned before, `awk` is primarily used for field based processing. Consider the sample input file shown below with fields separated by a single space character.


The learn_gnuawk repo has all the files used in examples.

```
$ cat table.txt
brown bread mat hair 42
blue cake mug shirt -7
yellow banana window shoes 3.14
```

Here's some examples that is based on specific field rather than entire line. By default, `awk` splits the input line based on spaces and the field contents can be accessed using `$N` where `N` is the field number required. A special variable `NF` is updated with the total number of fields for each input line. There's more details to cover, but for now this is enough to proceed.

```
$ # print the second field of each input line
$ awk '{print $2}' table.txt
bread
cake
banana

$ # print lines only if last field is a negative number
$ # recall that default action is to print contents of $0
$ awk '$NF<0' table.txt
blue cake mug shirt -7

$ # change 'b' to 'B' only for first field
$ awk '{gsub(/b/, "B", $1)} 1' table.txt
Brown bread mat hair 42
Blue cake mug shirt -7
yellow banana window shoes 3.14
```

awk one-liner structure

The examples in previous sections used a few different ways to construct a typical `awk` one-liner. If you haven't yet grasped the syntax, this generic structure might help:

```
awk 'cond1{action1} cond2{action2} ... condN{actionN}'
```

If a condition isn't provided, the action is always executed. Within a block, you can provide multiple statements separated by semicolon character. If action isn't provided, then by default, contents of `$0` variable is printed if the condition evaluates to `true` . When action isn't present, you can use semicolon to terminate a condition and start another `condX{actionX}` snippet.

Note that multiple blocks are just a syntactical sugar. It helps to avoid explicit use of `if` control structure for most one-liners. The below snippet shows the same code with and without `if` structure.

```
$ awk '{
         if($NF < 0){
            print $0
         }
       }' table.txt
blue cake mug shirt -7

$ awk '$NF<0' table.txt
blue cake mug shirt -7
```

You can use a `BEGIN{}` block when you need to execute something before input is read and a `END{}` block to execute something after all of the input has been processed.

```
$ seq 2 | awk 'BEGIN{print "---"} 1; END{print "%%%"}'
---
1
2
%%%
```

There are some more types of blocks that can be used, you'll see them in coming chapters. See gawk manual: Operators for details about operators and gawk manual: Truth Values and Conditions for conditional expressions.

Strings and Numbers

Some examples so far have already used string and numeric literals. As mentioned earlier, `awk` tries to provide a concise way to construct a solution from the command line. The data type of a value is determined based on the syntax used. String literals are represented inside double quotes. Numbers can be integers or floating point. Scientific notation is allowed as well. See gawk manual: Constant Expressions for more details.

```
$ # BEGIN{} is also useful to write awk program without any external input
$ awk 'BEGIN{print "hi"}'
hi
```

```
$ awk 'BEGIN{print 42}'
42
$ awk 'BEGIN{print 3.14}'
3.14
$ awk 'BEGIN{print 34.23e4}'
342300
```

You can also save these literals in variables and use it later. Some variables are predefined, for example `NF` .

```
$ awk 'BEGIN{a=5; b=2.5; print a+b}'
7.5

$ # strings placed next to each other are concatenated
$ awk 'BEGIN{s1="con"; s2="cat"; print s1 s2}'
concat
```

If uninitialized variable is used, it will act as empty string in string context and `0` in numeric context. You can force a string to behave as a number by simply using it in an expression with numeric values. You can also use unary `+` or `-` operators. If the string doesn't start with a valid number (ignoring any starting whitespaces), it will be treated as `0` . Similarly, concatenating a string to a number will automatically change the number to string. See gawk manual: How awk Converts Between Strings and Numbers for more details.

```
$ # same as: awk 'BEGIN{sum=0} {sum += $NF} END{print sum}'
$ awk '{sum += $NF} END{print sum}' table.txt
38.14

$ awk 'BEGIN{n1="5.0"; n2=5; if(n1==n2) print "equal"}'
$ awk 'BEGIN{n1="5.0"; n2=5; if(+n1==n2) print "equal"}'
equal
$ awk 'BEGIN{n1="5.0"; n2=5; if(n1==n2".0") print "equal"}'
equal

$ awk 'BEGIN{print 5 + "abc 2 xyz"}'
5
$ awk 'BEGIN{print 5 + " \t 2 xyz"}'
7
```

Arrays

Arrays in `awk` are associative, meaning they are key-value pairs. The keys can be both numbers or strings, but numbers get converted to strings internally. They can be multi-dimensional as well. There will be plenty of array examples in later chapters in relevant context. See gawk manual: Arrays for complete details and gotchas.

```
$ # assigning an array and accessing an element based on string key
$ awk 'BEGIN{student["id"] = 101; student["name"] = "Joe";
       print student["name"]}'
Joe
```

```
$ # checking if a key exists
$ awk 'BEGIN{student["id"] = 101; student["name"] = "Joe";
       if("id" in student) print "Key found"}'
Key found
```

Summary

In my early days of getting used to the Linux command line, I was intimidated by `sed` and `awk` examples and didn't even try to learn them. Hopefully, this gentler introduction works for you and the various syntactical magic has been explained adequately. Try to experiment with the given examples, for example change field number to something other than the number used. Be curious, like what happens if field number is negative or a floating-point number. Read the manual. Practice a lot.

Next chapter is dedicated solely for regular expressions. The features introduced in this chapter would be used in the examples, so make sure you are comfortable with `awk` syntax before proceeding.

Regular Expressions

Regular Expressions is a versatile tool for text processing. It helps to precisely define a matching criteria. For learning and understanding purposes, one can view regular expressions as a mini programming language in itself, specialized for text processing. Parts of a regular expression can be saved for future use, analogous to variables and functions. There are ways to perform AND, OR, NOT conditionals, features to concisely define repetition to avoid manual replication and so on.

Here's some common use cases.

- Sanitizing a string to ensure that it satisfies a known set of rules. For example, to check if a given string matches password rules.
- Filtering or extracting portions on an abstract level like alphabets, numbers, punctuation and so on.
- Qualified string replacement. For example, at the start or the end of a string, only whole words, based on surrounding text, etc.

This chapter will cover regular expressions as implemented in `awk`. Most of `awk`'s regular expression syntax is similar to Extended Regular Expression (ERE) found with `grep -E` and `sed -E`. Unless otherwise indicated, examples and descriptions will assume ASCII input.


See also POSIX specification for regular expressions. And unix.stackexchange: Why does my regular expression work in X but not in Y?

Syntax and variable assignment

As seen in previous chapter, the syntax is `string ~ /regexp/` to check if the given string satisfies the rules specified by the regexp. And `string !~ /regexp/` to invert the condition. By default, `$0` is checked if the string isn't specified. You can also save a regexp literal in a variable by prefixing `@` symbol. The prefix is needed because `/regexp/` by itself would mean `$0 ~ /regexp/`.

```
$ printf 'spared no one\ngrasped\nspar\n' | awk '/ed/'
spared no one
grasped

$ printf 'spared no one\ngrasped\nspar\n' | awk '{r = @/ed/} $0 ~ r'
spared no one
grasped
```

Line Anchors

In the examples seen so far, the regexp was a simple string value without any special characters. Also, the regexp pattern evaluated to `true` if it was found anywhere in the string. Instead of matching anywhere in the line, restrictions can be specified. These restrictions are made possible by assigning special meaning to certain characters and escape sequences. The characters with special meaning are known as **metacharacters** in regular expressions

parlance. In case you need to match those characters literally, you need to escape them with a `\` (discussed in Matching the metacharacters section).

There are two line anchors:

- `^` metacharacter restricts the matching to start of line
- `$` metacharacter restricts the matching to end of line

```
$ # lines starting with 'sp'
$ printf 'spared no one\ngrasped\nspar\n' | awk '/^sp/'
spared no one
spar

$ # lines ending with 'ar'
$ printf 'spared no one\ngrasped\nspar\n' | awk '/ar$/'
spar

$ # change only whole line 'spar'
$ # can also use: awk '/^spar$/{$0 = 123} 1'
$ printf 'spared no one\ngrasped\nspar\n' | awk '{sub(/^spar$/, "123")} 1'
spared no one
grasped
123
```

The anchors can be used by themselves as a pattern. Helps to insert text at start or end of line, emulating string concatenation operations. These might not feel like useful capability, but combined with other features they become quite a handy tool.

```
$ printf 'spared no one\ngrasped\nspar\n' | awk '{gsub(/^/, "* ")} 1'
* spared no one
* grasped
* spar

$ # append only if line doesn't contain space characters
$ printf 'spared no one\ngrasped\nspar\n' | awk '!/ /{gsub(/$/, ".")} 1'
spared no one
grasped.
spar.
```

Word Anchors

The second type of restriction is word anchors. A word character is any alphabet (irrespective of case), digit and the underscore character. You might wonder why there are digits and underscores as well, why not only alphabets? This comes from variable and function naming conventions — typically alphabets, digits and underscores are allowed. So, the definition is more programming oriented than natural language.

Use `\<` to indicate start of word anchor and `\>` to indicate end of word anchor. As an alternate, you can use `\y` to indicate both the start of word and end of word anchors.


Typically `\b` is used to represent word anchor (for example, in `grep`, `sed`,

`perl` , etc), but in `awk` the escape sequence `\b` always refers to the backspace character.

```
$ cat word_anchors.txt
sub par
spar
apparent effort
two spare computers
cart part tart mart

$ # words starting with 'par'
$ awk '/\<par/' word_anchors.txt
sub par
cart part tart mart

$ # words ending with 'par'
$ awk '/par\>/' word_anchors.txt
sub par
spar

$ # only whole word 'par'
$ # note that only lines where substitution succeeded will be printed
$ # as return value of sub/gsub is number of substitutions made
$ awk 'gsub(/\<par\>/, "***")' word_anchors.txt
sub ***
```


See also Word boundary differences section.

`\y` has an opposite too. `\B` matches locations other than those places where the word anchor would match.

```
$ # match 'par' if it is surrounded by word characters
$ awk '/\Bpar\B/' word_anchors.txt
apparent effort
two spare computers

$ # match 'par' but not as start of word
$ awk '/\Bpar/' word_anchors.txt
spar
apparent effort
two spare computers

$ # match 'par' but not as end of word
$ awk '/par\B/' word_anchors.txt
apparent effort
two spare computers
cart part tart mart
```

Here's an example for using word boundaries by themselves as a pattern. It also neatly shows the opposite functionality of `\y` and `\B` .

```
$ echo 'copper' | awk '{gsub(/\y/, ":")} 1'
:copper:
$ echo 'copper' | awk '{gsub(/\B/, ":")} 1'
c:o:p:p:e:r
```


Negative logic is handy in many text processing situations. But use it with care, you might end up matching things you didn't intend.

Combining conditions

Before seeing next regexp feature, it is good to note that sometimes using logical operators is easier to read and maintain compared to doing everything with regexp.

```
$ # lines starting with 'b' but not containing 'at'
$ awk '/^b/ && !/at/' table.txt
blue cake mug shirt -7

$ # if first field contains 'low' or last field is less than 0
$ awk '$1 ~ /low/ || $NF<0' table.txt
blue cake mug shirt -7
yellow banana window shoes 3.14
```

Alternation

Many a times, you'd want to search for multiple terms. In a conditional expression, you can use the logical operators to combine multiple conditions. With regular expressions, the `|` metacharacter is similar to logical OR. The regular expression will match if any of the expression separated by `|` is satisfied. These can have their own independent anchors as well.

Alternation is similar to using `||` operator between two regexps. Having a single regexp helps to write terser code and `||` cannot be used when substitution is required.

```
$ # lines with whole word 'par' or lines ending with 's'
$ # same as: awk '/\<par\>/ || /s$/'
$ awk '/\<par\>|s$/' word_anchors.txt
sub par
two spare computers

$ # replace 'cat' or 'dog' or 'fox' with '--'
$ echo 'cats dog bee parrot foxed' | awk '{gsub(/cat|dog|fox/, "--")} 1'
--s -- bee parrot --ed
```

There's some tricky situations when using alternation. If it is used for filtering a line, there is no ambiguity. However, for use cases like substitution, it depends on a few factors. Say, you want to replace `are` or `spared` — which one should get precedence? The bigger word `spared` or the substring `are` inside it or based on something else?

The alternative which matches earliest in the input gets precedence. Unlike other regular expression implementations, order of alternation doesn't affect the results.

```
$ # note that 'sub' is used here, so only first match gets replaced
$ echo 'cats dog bee parrot foxed' | awk '{sub(/bee|parrot|at/, "--")} 1'
c--s dog bee parrot foxed
$ echo 'cats dog bee parrot foxed' | awk '{sub(/parrot|at|bee/, "--")} 1'
c--s dog bee parrot foxed
```

In case of matches starting from same location, for example `spar` and `spared`, the longest matching portion gets precedence. See also Longest match wins section for more examples.

```
$ # example for alternations starting from same location
$ echo 'spared party parent' | awk '{sub(/spa|spared/, "**")} 1'
** party parent
$ echo 'spared party parent' | awk '{sub(/spared|spa/, "**")} 1'
** party parent
```

Grouping

Often, there are some common things among the regular expression alternatives. It could be common characters or qualifiers like the anchors. In such cases, you can group them using a pair of parentheses metacharacters. Similar to `a(b+c)d = abd+acd` in maths, you get `a(b|c)d = abd|acd` in regular expressions.

```
# without grouping
$ printf 'red\nreform\nread\narrest\n' | awk '/reform|rest/'
reform
arrest
# with grouping
$ printf 'red\nreform\nread\narrest\n' | awk '/re(form|st)/'
reform
arrest

# without grouping
$ printf 'sub par\nspare\npart time\n' | awk '/\<par\>|\<part\>/'
sub par
part time
# taking out common anchors
$ printf 'sub par\nspare\npart time\n' | awk '/\<(par|part)\>/'
sub par
part time
# taking out common characters as well
# you'll later learn a better technique instead of using empty alternate
$ printf 'sub par\nspare\npart time\n' | awk '/\<par(|t)\>/'
sub par
part time
```

Matching the metacharacters

You have seen a few metacharacters and escape sequences that help to compose a regular expression. To match the metacharacters literally, i.e. to remove their special meaning, prefix those characters with a `\` character. To indicate a literal `\` character, use `\\` .

Unlike `grep` and `sed` , the line anchors have to be always escaped to match them literally. They do not lose their special meaning when not used in their customary positions.

```
$ # awk '/b^2/' will not work even though ^ isn't being used as anchor
$ # however, b^2 will work for both grep and sed
$ echo 'a^2 + b^2 - C*3' | awk '/b\^2/'
a^2 + b^2 - C*3

$ # note that ')' doesn't need to be escaped
$ echo '(a*b) + c' | awk '{gsub(/\(|)/, "")} 1'
a*b + c

$ echo '\learn\by\example' | awk '{gsub(/\\/, "/")} 1'
/learn/by/example
```

Backreferences section will discuss how to handle the metacharacters in replacement section.

Using string literal as regexp

The first argument to `sub` and `gsub` functions can be a string as well, `awk` will handle converting it to a regexp. This has a few advantages. For example, if you have many `/` characters in the search pattern, it might become easier to use string instead of regexp.

```
$ p='/home/learnbyexample/reports'
$ echo "$p" | awk '{sub(/\/home\/learnbyexample\//, "~/")} 1'
~/reports
$ echo "$p" | awk '{sub("/home/learnbyexample/", "~/")} 1'
~/reports

$ # example with line matching instead of substitution
$ printf '/foo/bar/1\n/foo/baz/1\n' | awk '/\/foo\/bar\//'
/foo/bar/1
$ printf '/foo/bar/1\n/foo/baz/1\n' | awk '$0 ~ "/foo/bar/"'
/foo/bar/1
```

In the above examples, the string literal was supplied directly. But any other expression or variable can be used as well, examples for which will be shown later in this chapter. The reason why string isn't always used as the first argument is that the special meaning for `\` character will clash. For example:

```
$ awk 'gsub("\<par\>", "X")' word_anchors.txt
awk: cmd. line:1: warning: escape sequence `\<' treated as plain `<'
awk: cmd. line:1: warning: escape sequence `\>' treated as plain `>'
```

```
$ # you'll need \\ to represent \
$ awk 'gsub("\\<par\\>", "X")' word_anchors.txt
sub X
$ # much more readable with regexp literal
$ awk 'gsub(/\<par\>/, "X")' word_anchors.txt
sub X

$ # another example
$ echo '\learn\by\example' | awk '{gsub("\\\\", "/")} 1'
/learn/by/example
$ echo '\learn\by\example' | awk '{gsub(/\\/, "/")} 1'
/learn/by/example
```


See gawk manual: Gory details for more information than you'd want.

The dot meta character

The dot metacharacter serves as a placeholder to match any character (including newline character). Later you'll learn how to define your own custom placeholder for limited set of characters.

```
$ # 3 character sequence starting with 'c' and ending with 't'
$ echo 'tac tin cot abc:tyz excited' | awk '{gsub(/c.t/, "-")} 1'
ta-in - ab-yz ex-ed

$ # any character followed by 3 and again any character
$ printf '4\t35x\n' | awk '{gsub(/.3./, "")} 1'
4x

$ # 'c' followed by any character followed by 'x'
$ awk 'BEGIN{s="abc\nxyz"; sub(/c.x/, " ", s); print s}'
ab yz
```

Quantifiers

As an analogy, alternation provides logical OR. Combining the dot metacharacter `.` and quantifiers (and alternation if needed) paves a way to perform logical AND. For example, to check if a string matches two patterns with any number of characters in between. Quantifiers can be applied to both characters and groupings. Apart from ability to specify exact quantity and bounded range, these can also match unbounded varying quantities.

First up, the `?` metacharacter which quantifies a character or group to match `0` or `1` times. This helps to define optional patterns and build terser patterns compared to groupings for some cases.

```
$ # same as: awk '{gsub(/\<(fe.d|fed)\>/, "X")} 1'
$ echo 'fed fold fe:d feeder' | awk '{gsub(/\<fe.?d\>/, "X")} 1'
X fold X feeder
```

```
$ # same as: awk '/\<par(|t)\>/'
$ printf 'sub par\nspare\npart time\n' | awk '/\<part?\>/'
sub par
part time

$ # same as: awk '{gsub(/part|parrot/, "X")} 1'
$ echo 'par part parrot parent' | awk '{gsub(/par(ro)?t/, "X")} 1'
par X X parent
$ # same as: awk '{gsub(/part|parrot|parent/, "X")} 1'
$ echo 'par part parrot parent' | awk '{gsub(/par(en|ro)?t/, "X")} 1'
par X X X

$ # '<' to be replaced with '\<' only if not preceded by '\'
$ echo 'blah \< foo bar < blah baz <' | awk '{gsub(/\\?</, "\\<")} 1'
blah \< foo bar \< blah baz \<
```

The `*` metacharacter quantifies a character or group to match `0` or more times. There is no upper bound, more details will be discussed later in this section.

```
$ # 'f' followed by zero or more of 'e' followed by 'd'
$ echo 'fd fed fod fe:d feeeeder' | awk '{gsub(/fe*d/, "X")} 1'
X X fod fe:d Xer

$ # zero or more of '1' followed by '2'
$ echo '3111111111125111142' | awk '{gsub(/1*2/, "-")} 1'
3-511114-
```

The `+` metacharacter quantifies a character or group to match `1` or more times. Similar to `*` quantifier, there is no upper bound.

```
$ # 'f' followed by one or more of 'e' followed by 'd'
$ echo 'fd fed fod fe:d feeeeder' | awk '{gsub(/fe+d/, "X")} 1'
fd X fod fe:d Xer

$ # 'f' followed by at least one of 'e' or 'o' or ':' followed by 'd'
$ echo 'fd fed fod fe:d feeeeder' | awk '{gsub(/f(e|o|:)+d/, "X")} 1'
fd X X X Xer

$ # one or more of '1' followed by optional '4' and then '2'
$ echo '3111111111125111142' | awk '{gsub(/1+4?2/, "-")} 1'
3-5-
```

You can specify a range of integer numbers, both bounded and unbounded, using `{}` metacharacters. There are four ways to use this quantifier as listed below:

Pattern	Description
`{m,n}`	match `m` to `n` times
`{m,}`	match at least `m` times
`{,n}`	match up to `n` times (including `0` times)
`{n}`	match exactly `n` times

```
$ # note that inside {} space is not allowed
$ echo 'ac abc abbc abbbc abbbbbbbbc' | awk '{gsub(/ab{1,4}c/, "X")} 1'
ac X X X abbbbbbbbc

$ echo 'ac abc abbc abbbc abbbbbbbbc' | awk '{gsub(/ab{3,}c/, "X")} 1'
ac abc abbc X X

$ echo 'ac abc abbc abbbc abbbbbbbbc' | awk '{gsub(/ab{,2}c/, "X")} 1'
X X X abbbc abbbbbbbbc

$ echo 'ac abc abbc abbbc abbbbbbbbc' | awk '{gsub(/ab{3}c/, "X")} 1'
ac abc abbc X abbbbbbbbc
```

The `{}` metacharacters have to be escaped to match them literally. Similar to `()` metacharacters, escaping `{` alone is enough.

Next up, how to construct conditional AND using dot metacharacter and quantifiers.

```
$ # match 'Error' followed by zero or more characters followed by 'valid'
$ echo 'Error: not a valid input' | awk '/Error.*valid/'
Error: not a valid input
```

To allow matching in any order, you'll have to bring in alternation as well. But, for more than 3 patterns, the combinations become too many to write and maintain.

```
$ # 'cat' followed by 'dog' or 'dog' followed by 'cat'
$ echo 'two cats and a dog' | awk '{gsub(/cat.*dog|dog.*cat/, "pets")} 1'
two pets
$ echo 'two dogs and a cat' | awk '{gsub(/cat.*dog|dog.*cat/, "pets")} 1'
two pets
```

Longest match wins

You've already seen an example with alternation, where the longest matching portion was chosen if two alternatives started from same location. For example `spar|spared` will result in `spared` being chosen over `spar` . The same applies whenever there are two or more matching possibilities from same starting location. For example, `f.?o` will match `foo` instead of `fo` if the input string to match is `foot` .

```
$ # longest match among 'foo' and 'fo' wins here
$ echo 'foot' | awk '{sub(/f.?o/, "X")} 1'
Xt
$ # everything will match here
$ echo 'car bat cod map scat dot abacus' | awk '{sub(/.*/, "X")} 1'
X

$ # longest match happens when (1|2|3)+ matches up to '1233' only
$ # so that '12baz' can match as well
$ echo 'foo123312baz' | awk '{sub(/o(1|2|3)+(12baz)?/, "X")} 1'
foX
```

```
$ # in other implementations like 'perl', that is not the case
$ # quantifiers match as much as possible, but precedence is left to right
$ echo 'foo123312baz' | perl -pe 's/o(1|2|3)+(12baz)?/X/'
foXbaz
```

While determining the longest match, overall regular expression matching is also considered. That's how `Error.*valid` example worked. If `.*` had consumed everything after `Error`, there wouldn't be any more characters to try to match after `valid`. So, among the varying quantity of characters to match for `.*`, the longest portion that satisfies the overall regular expression is chosen. Something like `a.*b` will match from first `a` in the input string to the last `b` in the string. In other implementations, like `perl`, this is achieved through a process called **backtracking**. Both approaches have their own advantages and disadvantages and have cases where the regexp can result in exponential time consumption.

```
$ # from start of line to last 'm' in the line
$ echo 'car bat cod map scat dot abacus' | awk '{sub(/.*m/, "-")} 1'
-ap scat dot abacus

$ # from first 'b' to last 't' in the line
$ echo 'car bat cod map scat dot abacus' | awk '{sub(/b.*t/, "-")} 1'
car - abacus

$ # from first 'b' to last 'at' in the line
$ echo 'car bat cod map scat dot abacus' | awk '{sub(/b.*at/, "-")} 1'
car - dot abacus

$ # here 'm*' will match 'm' zero times as that gives the longest match
$ echo 'car bat cod map scat dot abacus' | awk '{sub(/a.*m*/, "-")} 1'
c-
```

Character classes

To create a custom placeholder for limited set of characters, enclose them inside `[]` metacharacters. It is similar to using single character alternations inside a grouping, but with added flexibility and features. Character classes have their own versions of metacharacters and provide special predefined sets for common use cases. Quantifiers are also applicable to character classes.

```
$ # same as: awk '/cot|cut/' and awk '/c(o|u)t/'
$ printf 'cute\ncat\ncot\ncoat\ncost\nscuttle\n' | awk '/c[ou]t/'
cute
cot
scuttle

$ # same as: awk '/.(a|e|o)+t/'
$ printf 'meeting\ncute\nboat\nat\nfoot\n' | awk '/.[aeo]+t/'
meeting
boat
foot
```

```
$ # same as: awk '{gsub(/\<(s|o|t)(o|n)\>/, "X")} 1'
$ echo 'no so in to do on' | awk '{gsub(/\<[sot][on]\>/, "X")} 1'
no X in X do X

$ # lines made up of letters 'o' and 'n', line length at least 2
$ # /usr/share/dict/words contains dictionary words, one word per line
$ awk '/^[on]{2,}$/' /usr/share/dict/words
no
non
noon
on
```

Character classes have their own metacharacters to help define the sets succinctly. Metacharacters outside of character classes like `^` , `$` , `()` etc either don't have special meaning or have completely different one inside the character classes.

First up, the `-` metacharacter that helps to define a range of characters instead of having to specify them all individually.

```
$ # same as: awk '{gsub(/[0123456789]+/, "-")} 1'
$ echo 'Sample123string42with777numbers' | awk '{gsub(/[0-9]+/, "-")} 1'
Sample-string-with-numbers

$ # whole words made up of lowercase alphabets and digits only
$ echo 'coat Bin food tar12 best' | awk '{gsub(/\<[a-z0-9]+\>/, "X")} 1'
X Bin X X X

$ # whole words made up of lowercase alphabets, starting with 'p' to 'z'
$ echo 'road i post grip read eat pit' | awk '{gsub(/\<[p-z][a-z]*\>/, "X")} 1'
X i X grip X eat X
```

Character classes can also be used to construct numeric ranges. However, it is easy to miss corner cases and some ranges are complicated to design. See also regular-expressions: Matching Numeric Ranges with a Regular Expression.

```
$ # numbers between 10 to 29
$ echo '23 154 12 26 34' | awk '{gsub(/\<[12][0-9]\>/, "X")} 1'
X 154 X X 34

$ # numbers >= 100 with optional leading zeros
$ echo '0501 035 154 12 26 98234' | awk '{gsub(/\<0*[1-9][0-9]{2,}\>/, "X")} 1'
X 035 X 12 26 X
```

Next metacharacter is `^` which has to specified as the first character of the character class. It negates the set of characters, so all characters other than those specified will be matched. Handle negative logic with care though, you might end up matching more than you wanted.

```
$ # replace all non-digits
$ echo 'Sample123string42with777numbers' | awk '{gsub(/[^0-9]+/, "-")} 1'
-123-42-777-

$ # delete last two columns based on a delimiter
```

```
$ echo 'foo:123:bar:baz' | awk '{sub(/(:[^:]+){2}$/, "")} 1'
foo:123

$ # sequence of characters surrounded by unique character
$ echo 'I like "mango" and "guava"' | awk '{gsub(/"[^"]+"/, "X")} 1'
I like X and X

$ # sometimes it is simpler to positively define a set than negation
$ # same as: awk '/^[^aeiou]*$/'
$ printf 'tryst\nfun\nglyph\npity\nwhy\n' | awk '!/[aeiou]/'
tryst
glyph
why
```

Some commonly used character sets have predefined escape sequences:

- `\w` matches all **word** characters `[a-zA-Z0-9_]` (recall the description for word boundaries)
- `\W` matches all non-word characters (recall duality seen earlier, like `\y` and `\B`)
- `\s` matches all **whitespace** characters: tab, newline, vertical tab, form feed, carriage return and space
- `\S` matches all non-whitespace characters

```
$ # match all non-word characters
$ echo 'load;err_msg--\/ant,r2..not' | awk '{gsub(/\W+/, "-")} 1'
load-err_msg-ant-r2-not

$ # replace all sequences of whitespaces with single space
$ printf 'hi  \v\f  there.\thave   \ra nice\t\tday\n' | awk '{gsub(/\s+/, " ")} 1'
hi there. have a nice day
```

These escape sequences *cannot* be used inside character classes.

```
$ # \w would simply match w inside character classes
$ echo 'w=y\x+9*3' | awk '{gsub(/[\w=]/, "")} 1'
y\x+9*3
```

`awk` doesn't support `\d` and `\D` , commonly featured in other implementations as a shortcut for all the digits and non-digits.

A **named character set** is defined by a name enclosed between `[:` and `:]` and has to be used within a character class `[]` , along with any other characters as needed.

Named set	Description
`[:digit:]`	`[0-9]`
`[:lower:]`	`[a-z]`
`[:upper:]`	`[A-Z]`
`[:alpha:]`	`[a-zA-Z]`
`[:alnum:]`	`[0-9a-zA-Z]`
`[:xdigit:]`	`[0-9a-fA-F]`
`[:cntrl:]`	control characters - first 32 ASCII characters and 127th (DEL)

Named set	Description
`[:punct:]`	all the punctuation characters
`[:graph:]`	`[:alnum:]` and `[:punct:]`
`[:print:]`	`[:alnum:]` , `[:punct:]` and space
`[:blank:]`	space and tab characters
`[:space:]`	whitespace characters, same as `\s`

```
$ s='err_msg xerox ant m_2 P2 load1 eel'
$ echo "$s" | awk '{gsub(/\<[[:lower:]]+\>/, "X")} 1'
err_msg X X m_2 P2 load1 X

$ echo "$s" | awk '{gsub(/\<[[:lower:]_]+\>/, "X")} 1'
X X X m_2 P2 load1 X

$ echo "$s" | awk '{gsub(/\<[[:alnum:]]+\>/, "X")} 1'
err_msg X X m_2 X X X

$ echo ',pie tie#ink-eat_42' | awk '{gsub(/[^[:punct:]]+/, "")} 1'
,#-_
```

Specific placement is needed to match character class metacharacters literally. Or, they can be escaped by prefixing `\` to avoid having to remember the different rules. As `\` is special inside character class, use `\\` to represent it literally.

```
$ # - should be first or last character within []
$ echo 'ab-cd gh-c 12-423' | awk '{gsub(/[a-z-]{2,}/, "X")} 1'
X X 12-423
$ # or escaped with \
$ echo 'ab-cd gh-c 12-423' | awk '{gsub(/[a-z\-0-9]{2,}/, "X")} 1'
X X X

$ # ] should be first character within []
$ printf 'int a[5]\nfoo\n1+1=2\n' | awk '/[=]]/'
$ printf 'int a[5]\nfoo\n1+1=2\n' | awk '/[]=]/'
int a[5]
1+1=2

$ # to match [ use [ anywhere in the character set
$ # [][] will match both [ and ]
$ printf 'int a[5]\nfoo\n1+1=2\n' | awk '/[][]/'
int a[5]

$ # ^ should be other than first character within []
$ echo 'f*(a^b) - 3*(a+b)/(a-b)' | awk '{gsub(/a[+^]b/, "c")} 1'
f*(c) - 3*(c)/(a-b)
```


Combinations like `[.` or `[:` cannot be used together to mean two individual characters, as they have special meaning within `[]` . See gawk manual: Using Bracket Expressions for more details.

```
$ echo 'int a[5]' | awk '/[x[.y]/'
awk: cmd. line:1: error: Unmatched [, [^, [:, [., or [=: /[x[.y]/
$ echo 'int a[5]' | awk '/[x[y.]/'
int a[5]
```

Escape sequences

Certain ASCII characters like tab `\t`, carriage return `\r`, newline `\n`, etc have escape sequences to represent them. Additionally, any character can be represented using their ASCII value in octal `\NNN` or hexadecimal `\xNN` formats. Unlike character set escape sequences like `\w`, these can be used inside character classes.

```
$ # using \t to represent tab character
$ printf 'foo\tbar\tbaz\n' | awk '{gsub(/\t/, " ")} 1'
foo bar baz

$ # these escape sequence work inside character class too
$ printf 'a\t\r\fb\vc\n' | awk '{gsub(/[\t\v\f\r]+/, ":")} 1'
a:b:c

$ # representing single quotes
$ # use \047 for octal format
$ echo "universe: '42'" | awk '{gsub(/\x27/, "")} 1'
universe: 42
```


See gawk manual: Escape Sequences for full list and other details.

Replace specific occurrence

The third substitution function is `gensub` which can be used instead of both `sub` and `gsub` functions. Syntax wise, `gensub` needs minimum three arguments. The third argument is used to indicate whether you want to replace all occurrences with `"g"` or specific occurrence by giving a number. Another difference is that `gensub` returns a string value (irrespective of substitution succeeding) instead of modifying the input.

```
$ # same as: sed 's/:/-/2'
$ # replace only second occurrence of ':' with '-'
$ # note that output of gensub is passed to print here
$ echo 'foo:123:bar:baz' | awk '{print gensub(/:/, "-", 2)}'
foo:123-bar:baz

$ # same as: sed -E 's/[^:]+/X/3'
$ # replace only third field with 'X'
$ echo 'foo:123:bar:baz' | awk '{print gensub(/[^:]+/, "X", 3)}'
foo:123:X:baz
```

The fourth argument for `gensub` function allows you to specify the input string or variable on which the substitution has to be performed. Default is `$0`, as seen in previous examples.

```
$ # replace vowels with 'X' only for fourth field
$ # same as: awk '{gsub(/[aeiou]/, "X", $4)} 1'
$ echo '1 good 2 apples' | awk '{$4 = gensub(/[aeiou]/, "X", "g", $4)} 1'
1 good 2 XpplXs
```

Backreferences

The grouping metacharacters `()` are also known as **capture groups**. They are like variables, the string captured by `()` can be referred later using backreference `\N` where `N` is the capture group you want. Leftmost `(` in the regular expression is `\1`, next one is `\2` and so on up to `\9`. As a special case, `\0` or `&` metacharacter represents entire matched string. As `\` is special inside double quotes, you'll have to use `"\\1"` to represent `\1`.

Backreferences of the form `\N` can only be used with `gensub` function. `&` can be used with `sub`, `gsub` and `gensub` functions.

```
$ # reduce \\ to single \ and delete if it is a single \
$ s='\[\] and \\w and \[a-zA-Z0-9\_\]'
$ echo "$s" | awk '{print gensub(/(\\?)\\/, "\\1", "g")}'
[] and \w and [a-zA-Z0-9_]

$ # duplicate first column value as final column
$ echo 'one,2,3.14,42' | awk '{print gensub(/^([^,]+).*/, "&,\\1", 1)}'
one,2,3.14,42,one

$ # add something at start and end of line
$ # as only '&' is used, gensub isn't needed here
$ echo 'hello world' | awk '{sub(/.*/, "Hi. &. Have a nice day")} 1'
Hi. hello world. Have a nice day

$ # here {N} refers to last but Nth occurrence
$ s='456:foo:123:bar:789:baz'
$ echo "$s" | awk '{print gensub(/(.*):((.*:){2})/, "\\1[]\\2", "g")}'
456:foo:123[]bar:789:baz
```

Unlike other regular expression implementations, like `grep` or `sed` or `perl`, backreferences cannot be used in search section in `awk`. See also unix.stackexchange: backreference in awk.

If quantifier is applied on a pattern grouped inside `()` metacharacters, you'll need an outer `()` group to capture the matching portion. Some regular expression engines provide non-capturing group to handle such cases. In `awk`, you'll have to work around the extra capture group.

```
$ # note the numbers used in replacement section
$ s='one,2,3.14,42'
$ echo "$s" | awk '{$0=gensub(/^(([^,]+,){2})([^,]+)/, "[\\1](\\3)", 1)} 1'
[one,2,](3.14),42
```

As `\` and `&` are special characters inside double quotes in replacement section, use `\\` and `\\&` respectively for literal representation.

```
$ echo 'foo and bar' | awk '{sub(/and/, "[&]")} 1'
foo [and] bar
$ echo 'foo and bar' | awk '{sub(/and/, "[\\&]")} 1'
foo [&] bar

$ echo 'foo and bar' | awk '{sub(/and/, "\\")} 1'
foo \ bar
```

Case insensitive matching

Unlike `sed` or `perl`, regular expressions in `awk` do not directly support the use of flags to change certain behaviors. For example, there is no flag to force the regexp to ignore case while matching.

The `IGNORECASE` special variable controls case sensitivity, which is `0` by default. By changing it to some other value (which would mean `true` in conditional expression), you can match case insensitively. The `-v` command line option allows you to assign a variable before input is read. The `BEGIN` block is also often used to change such settings.

```
$ printf 'Cat\ncOnCaT\nscatter\ncot\n' | awk -v IGNORECASE=1 '/cat/'
Cat
cOnCaT
scatter

$ # for small enough set, can also use character class
$ printf 'Cat\ncOnCaT\nscatter\ncot\n' | awk '{gsub(/[cC][aA][tT]/, "dog")} 1'
dog
cOndog
sdogter
cot
```

Another way is to use built-in string function `tolower` to change the input to lowercase first.

```
$ printf 'Cat\ncOnCaT\nscatter\ncot\n' | awk 'tolower($0) ~ /cat/'
Cat
cOnCaT
scatter
```

Dynamic regexp

As seen earlier, you can use a string literal instead of regexp to specify the pattern to be matched. Which implies that you can use any expression or a variable as well. This is helpful if you need to compute the regexp based on some conditions or if you are getting the pattern externally, such as user input.

The `-v` command line option comes in handy to get user input, say from a `bash` variable.

```
$ r='cat.*dog|dog.*cat'
$ echo 'two cats and a dog' | awk -v ip="$r" '{gsub(ip, "pets")} 1'
two pets

$ awk -v s='ow' '$0 ~ s' table.txt
brown bread mat hair 42
yellow banana window shoes 3.14

$ # you'll have to make sure to use \\ instead of \
$ r='\\<[12][0-9]\\>'
$ echo '23 154 12 26 34' | awk -v ip="$r" '{gsub(ip, "X")} 1'
X 154 X X 34
```

See Using shell variables chapter for a way to avoid having to use `\\` instead of `\` .

Sometimes, you need to get user input and then treat it literally instead of regexp pattern. In such cases, you'll need to first escape the metacharacters before using in substitution functions. Below example shows how to do it for search section. For replace section, you only have to escape the `\` and `&` characters.

```
$ awk -v s='(a.b)^{c}|d' 'BEGIN{gsub(/[{[(^$*?+.|\\]/, "\\\\&", s); print s}'
\(a\.b)\^\{c}\|d

$ echo 'f*(a^b) - 3*(a^b)' |
    awk -v s='(a^b)' '{gsub(/[{[(^$*?+.|\\]/, "\\\\&", s); gsub(s, "c")} 1'
f*c - 3*c

$ # match given input string literally, but only at end of line
$ echo 'f*(a^b) - 3*(a^b)' |
    awk -v s='(a^b)' '{gsub(/[{[(^$*?+.|\\]/, "\\\\&", s); gsub(s "$", "c")} 1'
f*(a^b) - 3*c
```

If you need to match instead of substitution, you can use the `index` function. See index section for details.

Summary

Regular expressions is a feature that you'll encounter in multiple command line programs and programming languages. It is a versatile tool for text processing. Although the features in `awk` are less compared to those found in programming languages, they are sufficient for most of the tasks you'll need for command line usage. It takes a lot of time to get used to syntax and features of regular expressions, so I'll encourage you to practice a lot and maintain notes. It'd also help to consider it as a mini-programming language in itself for its flexibility and complexity.

Field separators

Now that you are familiar with basic `awk` syntax and regular expressions, this chapter will dive deep into field processing. You'll learn how to set input and output field separators, how to use regexps for defining fields and how to work with fixed length fields.

Default field separation

As seen earlier, `awk` automatically splits input into fields which are accessible using `$N` where `N` is the field number you need. You can also pass an expression instead of numeric literal to specify the field required.

```
$ cat table.txt
brown bread mat hair 42
blue cake mug shirt -7
yellow banana window shoes 3.14

$ # print fourth field if first field starts with 'b'
$ awk '$1 ~ /^b/{print $4}' table.txt
hair
shirt

$ # print the field as specified by value stored in 'f' variable
$ awk -v f=3 '{print $f}' table.txt
mat
mug
window
```

The `NF` special variable will give you the number of fields for each input line. This is useful when you don't know how many fields are present in the input and you need to specify field number from the end.

```
$ # print the last field of each input line
$ awk '{print $NF}' table.txt
42
-7
3.14

$ # print the last but one field
$ awk '{print $(NF-1)}' table.txt
hair
shirt
shoes

$ # don't forget the parentheses!
$ awk '{print $NF-1}' table.txt
41
-8
2.14
```

By default, `awk` does more than split the input on spaces. It splits based on one or more sequence of space or tab or newline characters. In addition, any of these three characters at start or end of input gets trimmed and won't be part of field contents. Input containing newline character will be covered in Record separators chapter.

```
$ echo '   a   b   c   ' | awk '{print NF}'
3
$ # note that leading spaces isn't part of field content
$ echo '   a   b   c   ' | awk '{print $1}'
a
$ # note that trailing spaces isn't part of field content
$ echo '   a   b   c   ' | awk '{print $NF "."}'
c.

$ # here's another example with tab characters thrown in
$ printf '     one \t two\t\t\tthree  ' | awk '{print NF}'
3
$ printf '     one \t two\t\t\tthree  ' | awk '{print $2}'
two
```

When passing an expression for field number, floating-point result is acceptable too. The fractional portion is ignored. However, as precision is limited, it could result in rounding instead of truncation.

```
$ awk 'BEGIN{printf "%.16f\n", 2.999999999999999}'
2.9999999999999991
$ awk 'BEGIN{printf "%.16f\n", 2.9999999999999999}'
3.0000000000000000

$ # same as: awk '{print $2}' table.txt
$ awk '{print $2.999999999999999}' table.txt
bread
cake
banana
$ # same as: awk '{print $3}' table.txt
$ awk '{print $2.9999999999999999}' table.txt
mat
mug
window
```

Input field separator

The most common way to change the default field separator is to use the `-F` command line option. The value passed to the option will be treated as a regexp. For now, here's some examples without any special regexp characters.

```
$ # use ':' as input field separator
$ echo 'goal:amazing:whistle:kwality' | awk -F: '{print $1}'
goal
$ echo 'goal:amazing:whistle:kwality' | awk -F: '{print $NF}'
```

```
kwality

$ # use quotes to avoid clashes with shell special characters
$ echo 'one;two;three;four' | awk -F';' '{print $3}'
three

$ # first and last fields will have empty string as their values
$ echo '=a=b=c=' | awk -F= '{print $1 "," $NF "."}'
,.
```

You can also directly set the special `FS` variable to change the input field separator. This can be done from the command line using `-v` option or within the code blocks.

```
$ echo 'goal:amazing:whistle:kwality' | awk -v FS=: '{print $2}'
amazing

$ # field separator can be multiple characters too
$ echo '1e4SPT2k6SPT3a5SPT4z0' | awk 'BEGIN{FS="SPT"} {print $3}'
3a5
```

If you wish to split the input as individual characters, use an empty string as the field separator.

```
$ # note that the space between -F and '' is mandatory
$ echo 'apple' | awk -F '' '{print $1}'
a
$ echo 'apple' | awk -v FS= '{print $NF}'
e

$ # depending upon the locale, you can work with multibyte characters too
$ echo 'αλεπού' | awk -v FS= '{print $3}'
ε
```

Here's some examples with regexp field separator. The value passed to `-F` or `FS` is treated as a string and then converted to regexp. So, you'll need `\\` instead of `\` to mean a backslash character. The good news is that for single characters that are also regexp metacharacters, they'll be treated literally and you do not need to escape them.

```
$ echo 'Sample123string42with777numbers' | awk -F'[0-9]+' '{print $2}'
string
$ echo 'Sample123string42with777numbers' | awk -F'[a-zA-Z]+' '{print $2}'
123

$ # note the use of \\W to indicate \W
$ echo 'load;err_msg--\ant,r2..not' | awk -F'\\W+' '{print $3}'
ant

$ # same as: awk -F'\\.' '{print $2}'
$ echo 'hi.bye.hello' | awk -F. '{print $2}'
bye

$ # count number of vowels for each input line
$ printf 'cool\nnice car\n' | awk -F'[aeiou]' '{print NF-1}'
```

```
2
3
```


The default value of `FS` is single space character. So, if you set input field separator to single space, then it will be the same as if you are using the default split discussed in previous section. If you want to override this behavior, you can use space inside a character class.

```
$ # same as: awk '{print NF}'
$ echo '   a   b   c   ' | awk -F' ' '{print NF}'
3

$ # there are 12 space characters, thus 13 fields
$ echo '   a   b   c   ' | awk -F'[ ]' '{print NF}'
13
```

Output field separator

The `OFS` special variable is used for output field separator. `OFS` is used as the string between multiple arguments passed to `print` function. It is also used whenever `$0` has to be reconstructed as a result of changing field contents. The default value for `OFS` is a single space character, just like for `FS` . There is no command line option though, you'll have to change `OFS` directly.

```
$ # printing first and third field, OFS is used to join these values
$ # note the use of , to separate print arguments
$ awk '{print $1, $3}' table.txt
brown mat
blue mug
yellow window

$ # same FS and OFS
$ echo 'goal:amazing:whistle:kwality' | awk -F: -v OFS=: '{print $2, $NF}'
amazing:kwality
$ echo 'goal:amazing:whistle:kwality' | awk 'BEGIN{FS=OFS=":"} {print $2, $NF}'
amazing:kwality

$ # different values for FS and OFS
$ echo 'goal:amazing:whistle:kwality' | awk -F: -v OFS=- '{print $2, $NF}'
amazing-kwality
```

Here's some examples for changing field contents and then printing `$0` .

```
$ echo 'goal:amazing:whistle:kwality' | awk -F: -v OFS=: '{$2 = 42} 1'
goal:42:whistle:kwality
$ echo 'goal:amazing:whistle:kwality' | awk -F: -v OFS=, '{$2 = 42} 1'
goal,42,whistle,kwality

$ # recall that spaces at start/end gets trimmed for default FS
$ echo '   a   b   c   ' | awk '{$NF = "last"} 1'
```

```
a b last
```

Sometimes you want to print contents of `$0` with the new `OFS` value but field contents aren't being changed. In such cases, you can assign a field value to itself to force reconstruction of `$0` .

```
$ # no change because there was no trigger to rebuild $0
$ echo 'Sample123string42with777numbers' | awk -F'[0-9]+' -v OFS=, '1'
Sample123string42with777numbers

$ # assign a field to itself in such cases
$ echo 'Sample123string42with777numbers' | awk -F'[0-9]+' -v OFS=, '{$1=$1} 1'
Sample,string,with,numbers
```

Manipulating NF

Changing `NF` value will rebuild `$0` as well.

```
$ # reducing fields
$ echo 'goal:amazing:whistle:kwality' | awk -F: -v OFS=, '{NF=2} 1'
goal,amazing

$ # increasing fields
$ echo 'goal:amazing:whistle:kwality' | awk -F: -v OFS=: '{$(NF+1)="sea"} 1'
goal:amazing:whistle:kwality:sea

$ # empty fields will be created as needed
$ echo 'goal:amazing:whistle:kwality' | awk -F: -v OFS=: '{$8="go"} 1'
goal:amazing:whistle:kwality::::go
```

Assigning `NF` to `0` will delete all the fields. However, a negative value will result in an error.

```
$ echo 'goal:amazing:whistle:kwality' | awk -F: -v OFS=: '{NF=-1} 1'
awk: cmd. line:1: (FILENAME=- FNR=1) fatal: NF set to negative value
```

FPAT

`FS` allows to define input field separator. In contrast, `FPAT` (field pattern) allows to define what should the fields be made up of.

```
$ s='Sample123string42with777numbers'

$ # define fields to be one or more consecutive digits
$ echo "$s" | awk -v FPAT='[0-9]+' '{print $2}'
42

$ # define fields to be one or more consecutive alphabets
```

```
$ echo "$s" | awk -v FPAT='[a-zA-Z]+' -v OFS=, '{$1=$1} 1'
Sample,string,with,numbers
```

`FPAT` is often used for `csv` input where fields can contain embedded delimiter characters. For example, a field content `"fox,42"` when `,` is the delimiter.

```
$ s='eagle,"fox,42",bee,frog'

$ # simply using , as separator isn't sufficient
$ echo "$s" | awk -F, '{print $2}'
"fox
```

For such simpler `csv` input, `FPAT` helps to define fields as starting and ending with double quotes or containing non-comma characters.

```
$ # * is used instead of + to allow empty fields
$ echo "$s" | awk -v FPAT='"[^"]*"|[^,]*' '{print $2}'
"fox,42"
```

The above will not work for all kinds of `csv` files, for example if fields contain escaped double quotes, newline characters, etc. See stackoverflow: What's the most robust way to efficiently parse CSV using awk? for such cases. You could also use other programming languages such as Perl, Python, Ruby, etc which come with standard `csv` parsing libraries or have easy access to third party solutions. There are also specialized command line tools such as xsv.

FIELDWIDTHS

`FIELDWIDTHS` is another feature where you get to define field contents. As indicated by the name, you have specify number of characters for each field. This method is useful to process fixed width file inputs, and especially when they can contain empty fields.

```
$ cat items.txt
apple   fig banana
50      10  200

$ # here field widths have been assigned such that
$ # extra spaces are placed at the end of each field
$ awk -v FIELDWIDTHS='8 4 6' '{print $2}' items.txt
fig
10
$ # note that the field contents will include the spaces as well
$ awk -v FIELDWIDTHS='8 4 6' '{print "[" $2 "]"}' items.txt
[fig ]
[10  ]
```

You can optionally prefix a field width with number of characters to be ignored.

```
$ # first field is 5 characters
$ # then 3 characters are ignored and 3 characters for second field
$ # then 1 character is ignored and 6 characters for third field
```

```
$ awk -v FIELDWIDTHS='5 3:3 1:6' '{print "[" $1 "]"}' items.txt
[apple]
[50   ]
$ awk -v FIELDWIDTHS='5 3:3 1:6' '{print "[" $2 "]"}' items.txt
[fig]
[10 ]
```

If an input line length exceeds the total widths specified, the extra characters will simply be ignored. If you wish to access those characters, you can use `*` to represent the last field. See gawk manual: FIELDWIDTHS for more corner cases.

```
$ awk -v FIELDWIDTHS='5 *' '{print "[" $1 "]"}' items.txt
[apple]
[50   ]

$ awk -v FIELDWIDTHS='5 *' '{print "[" $2 "]"}' items.txt
[   fig banana]
[   10  200]
```

Summary

Working with fields is the most popular feature of `awk` . This chapter discussed various ways in which you can split the input into fields and manipulate them. There's many more examples to be discussed related to fields in upcoming chapters. I'd highly suggest to also read through gawk manual: Fields for more details regarding field processing.

Next chapter will discuss various ways to use record separators and related special variables.

Record separators

So far, you've seen examples where `awk` automatically splits input line by line based on the `\n` newline character. Just like you can control how those lines are further split into fields using `FS` and other features, `awk` provides a way to control what constitutes a line in the first place. In `awk` parlance, the term **record** is used to describe the contents that gets placed in the `$0` variable. And similar to `OFS`, you can control the string that gets added at the end for `print` function. This chapter will also discuss how you can use special variables that have information related to record (line) numbers.

Input record separator

The `RS` special variable is used to control how the input content is split into records. The default is `\n` newline character, as evident with examples used in previous chapters. The special variable `NR` keeps track of the current record number.

```
$ # changing input record separator to comma
$ # note the content of second record, newline is just another character
$ printf 'this,is\na,sample' | awk -v RS=, '{print NR ")", $0}'
1) this
2) is
a
3) sample
```

Recall that default `FS` will split input record based on spaces, tabs and newlines. Now that you've seen how `RS` can be something other than newline, here's an example to show the full effect of default record splitting.

```
$ s='   a\t\tb:1000\n\n\n\n123 7777:x  y \n \n z  '
$ printf '%b' "$s" | awk -v RS=: -v OFS=, '{$1=$1} 1'
a,b
1000,123,7777
x,y,z
```

The `RS` value is treated as a regexp, just like it did for `FS`. For now, consider an example with multiple characters for `RS` but without needing regexp metacharacters.

```
$ cat report.log
blah blah Error: second record starts
something went wrong
some more details Error: third record
details about what went wrong

$ # uses 'Error:' as the input record separator
$ # prints all the records that contains 'something'
$ awk -v RS='Error:' '/something/' report.log
 second record starts
something went wrong
some more details
```

The default line ending for text files varies between different platforms. For example, a text file downloaded from internet or a file originating from Windows OS would typically have lines ending with carriage return and line feed characters. So, you'll have to use `RS='\r\n'` for such files. See also stackoverflow: Why does my tool output overwrite itself and how do I fix it? for a detailed discussion and mitigation methods.

Output record separator

The `ORS` special variable is used for output record separator. `ORS` is the string that gets added to the end of every call to the `print` function. The default value for `ORS` is a single newline character, just like `RS` .

```
$ # change NUL record separator to dot and newline
$ printf 'foo\0bar\0' | awk -v RS='\0' -v ORS='.\n' '1'
foo.
bar.

$ cat msg.txt
Hello there.
It will rain to-
day. Have a safe
and pleasant jou-
rney.
$ # here ORS is empty string
$ awk -v RS='-\n' -v ORS= '1' msg.txt
Hello there.
It will rain today. Have a safe
and pleasant journey.
```

Note that the `$0` variable is assigned after removing trailing characters matched by `RS` . Thus, you cannot directly manipulate those characters with functions like `sub` . With tools that don't automatically strip record separator, such as `perl` , the previous example can be solved as `perl -pe 's/-\n//' msg.txt` .

Many a times, you need to change `ORS` depending upon contents of input record or some other condition. The `cond ? expr1 : expr2` ternary operator is often used in such scenarios. The below example assumes that input is evenly divisible, you'll have to add more logic if that is not the case.

```
$ # can also use RS instead of "\n" here
$ seq 6 | awk '{ORS = NR%3 ? "-" : "\n"} 1'
1-2-3
4-5-6
```

If the last line of input didn't end with the input record separator, it might get added in the output if `print` is used, as `ORS` gets appended.

```
$ # here last line of input didn't end with newline
$ # but gets added via ORS when 'print' is used
```

```
$ printf '1\n2' | awk '1; END{print 3}'
1
2
3
```

Regexp RS and RT

As mentioned before, the value passed to `RS` is treated as a regular expression. Here's some examples.

```
$ # set input record separator as one or more digit characters
$ # print records containing 'i' and 't'
$ printf 'Sample123string42with777numbers' | awk -v RS='[0-9]+' '/i/ && /t/'
string
with

$ # similar to FS, the value passed to RS is string literal
$ # which is then converted to regexp, so need \\ instead of \ here
$ printf 'load;err_msg--ant,r2..not' | awk -v RS='\\W+' '/an/'
ant
```

First record will be empty if `RS` matches from the start of input file. However, if `RS` matches until the very last character of the input file, there won't be empty record as the last record. This is different from how `FS` behaves if it matches until the last character.

```
$ # first record is empty and last record is newline character
$ # change 'echo' command to 'printf' and see what changes
$ echo '123string42with777' | awk -v RS='[0-9]+' '{print NR ") [" $0 "]"}'
1) []
2) [string]
3) [with]
4) [
]

$ printf '123string42with777' | awk -v FS='[0-9]+' '{print NF}'
4
$ printf '123string42with777' | awk -v RS='[0-9]+' 'END{print NR}'
3
```

The `RT` special variable contains the text that was matched by `RS` . This variable gets updated for every input record.

```
$ # print record number and value of RT for that record
$ # last record has empty RT because it didn't end with digits
$ echo 'Sample123string42with777numbers' | awk -v RS='[0-9]+' '{print NR, RT}'
1 123
2 42
3 777
4
```

Paragraph mode

As a special case, when `RS` is set to empty string, one or more consecutive empty lines is used as the input record separator. Consider the below sample file:

```
$ cat programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,
by definition, not smart enough to debug it by Brian W. Kernighan

Some people, when confronted with a problem, think - I know, I will
use regular expressions. Now they have two problems by Jamie Zawinski

A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis

There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick
```

Here's an example of processing input paragraph wise.

```
$ # print all paragraphs containing 'you'
$ # note that there'll be an empty line after the last record
$ awk -v RS= -v ORS='\n\n' '/you/' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,
by definition, not smart enough to debug it by Brian W. Kernighan

A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis

```

The empty line at the end is a common problem when dealing with custom record separators. You could either process the output to remove it or add logic to avoid the extras. Here's one workaround for the previous example.

```
$ # here ORS is left as default newline character
$ # uninitialized variable will be false in conditional expression
$ # after the first record is printed, counter 'c' becomes non-zero
$ awk -v RS= '/you/{print c++ ? "\n" $0 : $0}' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,
by definition, not smart enough to debug it by Brian W. Kernighan

A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis
```

Paragraph mode is not the same as using `RS='\n\n+'` because `awk` does a few more operations when `RS` is empty. See gawk manual: multiline records for details. Important points are quoted below and illustrated with examples.

> However, there is an important difference between `RS = ""` and `RS = "\n\n+"` . In the first case, leading newlines in the input data file are ignored

```
$ s='\n\n\na\nb\n\n12\n34\n\nhi\nhello\n'

$ # paragraph mode
$ printf '%b' "$s" | awk -v RS= -v ORS='\n---\n' 'NR<=2'
a
b
---
12
34
---

$ # RS is '\n\n+' instead of paragraph mode
$ printf '%b' "$s" | awk -v RS='\n\n+' -v ORS='\n---\n' 'NR<=2'

---
a
b
---
```

and if a file ends without extra blank lines after the last record, the final newline is removed from the record. In the second case, this special processing is not done.

```
$ s='\n\n\na\nb\n\n12\n34\n\nhi\nhello\n'

$ # paragraph mode
$ printf '%b' "$s" | awk -v RS= -v ORS='\n---\n' 'END{print}'
hi
hello
---

$ # RS is '\n\n+' instead of paragraph mode
$ printf '%b' "$s" | awk -v RS='\n\n+' -v ORS='\n---\n' 'END{print}'
hi
hello

---
```

When RS is set to the empty string and FS is set to a single character, the newline character always acts as a field separator. This is in addition to whatever field separations result from FS. When FS is the null string (`""`) or a regexp, this special feature of RS does not apply. It does apply to the default field separator of a single space: `FS = " "` .

```
$ s='a:b\nc:d\n\n1\n2\n3'

$ # FS is a single character in paragraph mode
$ printf '%b' "$s" | awk -F: -v RS= -v ORS='\n---\n' '{$1=$1} 1'
a b c d
---
1 2 3
---
```

```
$ # FS is a regexp in paragraph mode
$ printf '%b' "$s" | awk -F':+' -v RS= -v ORS='\n---\n' '{$1=$1} 1'
a b
c d
---
1
2
3
---

$ # FS is single character and RS is '\n\n+' instead of paragraph mode
$ printf '%b' "$s" | awk -F: -v RS='\n\n+' -v ORS='\n---\n' '{$1=$1} 1'
a b
c d
---
1
2
3
---
```

NR vs FNR

There are two special variables related to record number. You've seen `NR` earlier in the chapter, but here's some more examples.

```
$ # same as: head -n2
$ seq 5 | awk 'NR<=2'
1
2

$ # same as: tail -n1
$ awk 'END{print}' table.txt
yellow banana window shoes 3.14

$ # change first field content only for second line
$ awk 'NR==2{$1="green"} 1' table.txt
brown bread mat hair 42
green cake mug shirt -7
yellow banana window shoes 3.14
```

All the examples with `NR` so far has been with single file input. If there are multiple file inputs, then you can choose between `NR` and the second special variable `FNR`. The difference is that `NR` contains total records read so far whereas `FNR` contains record number of only the current file being processed. Here's some examples to show them in action. You'll see more examples in later chapters as well.

```
$ awk -v OFS='\t' 'BEGIN{print "NR", "FNR", "Content"}
                   {print NR, FNR, $0}' report.log table.txt
NR      FNR     Content
```

```
1       1       blah blah Error: second record starts
2       2       something went wrong
3       3       some more details Error: third record
4       4       details about what went wrong
5       1       brown bread mat hair 42
6       2       blue cake mug shirt -7
7       3       yellow banana window shoes 3.14

$ # same as: head -q -n1
$ awk 'FNR==1' report.log table.txt
blah blah Error: second record starts
brown bread mat hair 42
```


For large input files, use `exit` to avoid unnecessary record processing.

```
$ seq 3542 4623452 | awk 'NR==2452{print; exit}'
5993
$ seq 3542 4623452 | awk 'NR==250; NR==2452{print; exit}'
3791
5993

$ # here is a sample time comparison
$ time seq 3542 4623452 | awk 'NR==2452{print; exit}' > f1
real    0m0.004s
$ time seq 3542 4623452 | awk 'NR==2452' > f2
real    0m0.717s
```

Summary

This chapter showed you how to change the way input content is split into records and how to set the string to be appended when `print` is used. The paragraph mode is useful for processing multiline records separated by empty lines. You also learned two special variables related to record numbers and where to use them.

So far, you've used `awk` to manipulate file content without modifying the source file. The next chapter will discuss how to write back the changes to the original input files.

In-place file editing

In the examples presented so far, the output from `awk` was displayed on the terminal. This chapter will discuss how to write back the changes to the input file(s) itself using the `-i` command line option. This option can be configured to make changes to the input file(s) with or without creating a backup of original contents.

Without backup

The `-i` option allows you to load libraries (see documentation for details). `inplace` library comes by default with the `awk` installation. Use `-i inplace` to indicate that `awk` should modify the original input itself. Use this option with caution, preferably after testing that the code is working as intended.

```
$ cat greeting.txt
Hi there
Have a nice day
Good bye

$ # prefix line numbers
$ awk -i inplace '{print NR ". " $0}' greeting.txt
$ cat greeting.txt
1. Hi there
2. Have a nice day
3. Good bye
```

Multiple input files are treated individually and changes are written back to respective files.

```
$ cat f1.txt
I ate 3 apples
$ cat f2.txt
I bought two balls and 3 bats

$ awk -i inplace '{gsub(/\<3\>/, "three")} 1' f1.txt f2.txt
$ cat f1.txt
I ate three apples
$ cat f2.txt
I bought two balls and three bats
```

With backup

You can provide a backup extension by setting the `inplace::suffix` special variable. For example, if the input file is `ip.txt` and `inplace::suffix='.orig'` is used, the backup file will be named as `ip.txt.orig` .

```
$ cat f3.txt
  Name    Physics  Maths
 Moe  76  82
```

```
Raj  56  64

$ awk -i inplace -v inplace::suffix='.bkp' -v OFS=, '{$1=$1} 1' f3.txt
$ cat f3.txt
Name,Physics,Maths
Moe,76,82
Raj,56,64

$ # original file is preserved in 'f3.txt.bkp'
$ cat f3.txt.bkp
  Name    Physics  Maths
 Moe  76  82
Raj  56  64
```

Earlier versions of `awk` used `INPLACE_SUFFIX` variable instead of `inplace::suffix` . Also, you can use `inplace::enable` variable to dynamically control whether files should be in-placed or not. See gawk manual: Enabling In-Place File Editing for more details.

Summary

This chapter discussed about the `-i inplace` option which is useful when you need to edit a file in-place. This is particularly useful in automation scripts. But, do ensure that you have tested the `awk` command before applying to actual files if you need to use this option without creating backups.

The next chapter will revisit the use of shell variables in `awk` commands.

Using shell variables

When it comes to automation and scripting, you'd often need to construct commands that can accept input from user, file, output of a shell command, etc. As mentioned before, this book assumes `bash` as the shell being used.

As an example, see my repo ch: command help for a practical shell script, where commands are constructed dynamically.

-v option

The most common method is to use the `-v` command line option.

```
$ # assume that the 's' variable is part of some bash script
$ # or perhaps a variable that has stored the output of a shell command
$ s='cake'
$ awk -v word="$s" '$2==word' table.txt
blue cake mug shirt -7
```

ENVIRON

To access environment variables of the shell, you can call the special array variable `ENVIRON` with the name of the environment variable as a string key.

```
$ # existing environment variable
$ # output shown here is for my machine, would differ for you
$ awk 'BEGIN{print ENVIRON["HOME"]}'
/home/learnbyexample
$ awk 'BEGIN{print ENVIRON["SHELL"]}'
/bin/bash

$ # defined along with awk command
$ # note that the variable is placed before awk
$ word='hello' awk 'BEGIN{print ENVIRON["word"]}'
hello
```

`ENVIRON` is a good way to get around `awk` 's interpretation of escape sequences. This is especially helpful for fixed string matching, see index section for examples.

```
$ s='hi\nbye'

$ # when passed via -v option
$ awk -v ip="$s" 'BEGIN{print ip}'
hi
bye

$ # when passed as an environment variable
$ ip="$s" awk 'BEGIN{print ENVIRON["ip"]}'
hi\nbye
```

Here's another example when a regexp is passed to an `awk` command.

```
$ # when passed via -v option
$ r='\Bpar\B'
$ awk -v rgx="$r" '$0 ~ rgx' word_anchors.txt
awk: warning: escape sequence `\B' treated as plain `B'
$ r='\\Bpar\\B'
$ awk -v rgx="$r" '$0 ~ rgx' word_anchors.txt
apparent effort
two spare computers

$ # when passed as an environment variable
$ r='\Bpar\B'
$ rgx="$r" awk '$0 ~ ENVIRON["rgx"]' word_anchors.txt
apparent effort
two spare computers
```

Summary

This short chapter revisited the `-v` command line option and introduced the `ENVIRON` special array. These are particularly useful when the `awk` command is part of a shell script. More about arrays will be discussed in later chapters.

The next chapter will cover control structures.

Control Structures

You've already seen various examples with conditional expressions playing a role in solving the problem. This chapter will revisit `if-else` control structure along with the ternary operator. Then you will see some examples with explicit loops (recall that `awk` is already looping over input records). Followed by keywords that control loop flow. Most of the syntax is very similar to the `C` language.

if-else

Mostly, when you need to use `if` control structure, you can get away with using `condX{actionX}` format in `awk` one-liners. But sometimes, you need additional condition checking within such action blocks. Or, you might need it inside loops. The syntax is `if(cond){action}` where the braces are optional if you need only one statement. `if` can be optionally followed by multiple `else if` conditions and a final `else` condition. These can also be nested as needed.

```
$ # print all lines starting with 'b'
$ # additionally, if last column is > 0, then print some more info
$ awk '/^b/{print; if($NF>0) print "------"}' table.txt
brown bread mat hair 42
------
blue cake mug shirt -7

$ # same as above, but includes 'else' condition as well
$ awk '/^b/{print; if($NF>0) print "------"; else print "======"}' table.txt
brown bread mat hair 42
------
blue cake mug shirt -7
======
```

The ternary operator often reduces the need for single statement `if-else` cases.

```
$ # same as: awk '{if(NR%3) ORS="-" ; else ORS=RS} 1'
$ seq 6 | awk '{ORS = NR%3 ? "-" : RS} 1'
1-2-3
4-5-6

$ # note that parentheses is necessary for print in this case
$ awk '/^b/{print; print($NF>0 ? "------" : "======")}' table.txt
brown bread mat hair 42
------
blue cake mug shirt -7
======
```


See also gawk manual: switch.

loops

`for` loops are handy when you are working with arrays. Since input fields can be referred dynamically by passing an expression with `$N` syntax, `for` loops are useful for processing them as well.

```
$ awk 'BEGIN{for(i=2; i<7; i+=2) print i}'
2
4
6

$ # looping each field
$ awk -v OFS=, '{for(i=1; i<=NF; i++) if($i ~ /^[bm]/) $i="["$i"]"} 1' table.txt
[brown],[bread],[mat],hair,42
[blue],cake,[mug],shirt,-7
yellow,[banana],window,shoes,3.14
```

Here's an example of looping over arrays where the keys do not follow any numerical arithmetic progression.

```
$ cat marks.txt
Dept    Name    Marks
ECE     Raj     53
ECE     Joel    72
EEE     Moi     68
CSE     Surya   81
EEE     Tia     59
ECE     Om      92
CSE     Amy     67

$ # average marks for each department
$ awk 'NR>1{d[$1]+=$3; c[$1]++} END{for(k in d) print k, d[k]/c[k]}' marks.txt
ECE 72.3333
EEE 63.5
CSE 74
```

You can use `break` and `continue` to alter the normal flow of loops. `break` will cause the current loop to quit immediately without processing the remaining statements and iterations. `continue` will skip the remaining statements in the loop and start next iteration.

```
$ awk -v OFS=, '{for(i=1; i<=NF; i++) if($i ~ /b/){NF=i; break}} 1' table.txt
brown
blue
yellow,banana
```

`awk` also supports `while` and `do-while` loop mechanisms.

```
$ awk 'BEGIN{i=6; while(i>0){print i; i-=2}}'
6
4
2

$ # recursive substitution
```

```
$ echo 'titillate' | awk '{while(gsub(/til/, "")) print}'
tilate
ate
```

next

`next` is similar to `continue` statement but it acts on the default loop that goes through the input records. It doesn't affect `BEGIN` or `END` blocks as they are outside the record looping. When `next` is executed, rest of the statements will be skipped and next input record will be fetched for processing.

```
$ awk '/\<par/{print "%% " $0; next} {print /s/ ? "X" : "Y"}' word_anchors.txt
%% sub par
X
Y
X
%% cart part tart mart
```

You'll see more examples with `next` in coming chapters.

exit

You saw the use of `exit` earlier to quit early and avoid unnecessary processing of records. If an argument isn't passed, `awk` considers the command to have finished normally and **exit status** will indicate success. You can pass a number to indicate other cases.

```
$ seq 3542 4623452 | awk 'NR==2452{print; exit}'
5993
$ echo $?
0

$ awk '/^br/{print "Invalid input"; exit 1}' table.txt
Invalid input
$ echo $?
1

$ # any remaining files to be processed are also skipped
$ awk 'FNR==2{print; exit}' table.txt greeting.txt
blue cake mug shirt -7
```

If `exit` is used in `BEGIN` or normal blocks, any code in `END` block will still be executed. For more details and corner cases, see gawk manual: exit.

```
$ # first print is executed
$ # on seeing exit, rest of BEGIN and normal blocks are skipped
$ # code in END is then executed
$ awk 'BEGIN{print "hi"; exit; print "hello"}
       /^b/;
       END{print "bye"}' table.txt
```

```
hi
bye
```

Summary

This chapter covered some of the control flow structures provided by `awk` . These features makes `awk` flexible and easier to use compared to `sed` for those familiar with programming languages.

Next chapter will discuss some of the built-in functions.

Built-in functions

You've already seen some built-in functions in detail, such as `sub` , `gsub` and `gensub` functions. This chapter will discuss many more built-ins that are often used in one-liners. You'll also see arrays in action.

See gawk manual: Functions for details about all the built-in functions as well as how to define your own functions.

length

`length` function returns number of characters for the given string argument. By default, it acts on `$0` variable and a number argument is converted to string automatically.

```
$ awk 'BEGIN{print length("road"); print length(123456)}'
4
6

$ # recall that record separator isn't part of $0
$ # so, line ending won't be counted here
$ printf 'fox\ntiger\n' | awk '{print length()}'
3
5

$ awk 'length($1) < 6' table.txt
brown bread mat hair 42
blue cake mug shirt -7
```

If you need number of bytes, instead of number of characters, then use the `-b` command line option as well.

```
$ echo 'αλεπού' | awk '{print length()}'
6
$ echo 'αλεπού' | awk -b '{print length()}'
12
```

Array sorting

By default, array looping with `for(key in array)` format gives you elements in random order. By setting a special value to `PROCINFO["sorted_in"]` , you can control the order in which you wish to retrieve the elements. See gawk manual: Using Predefined Array Scanning Orders for other options and details.

```
$ # by default, array is traversed in random order
$ awk 'BEGIN{a["z"]=1; a["x"]=12; a["b"]=42; for(i in a) print i, a[i]}'
x 12
z 1
b 42
```

```
$ # index sorted ascending order as strings
$ awk 'BEGIN{PROCINFO["sorted_in"] = "@ind_str_asc";
       a["z"]=1; a["x"]=12; a["b"]=42; for(i in a) print i, a[i]}'
b 42
x 12
z 1

$ # value sorted ascending order as numbers
$ awk 'BEGIN{PROCINFO["sorted_in"] = "@val_num_asc";
       a["z"]=1; a["x"]=12; a["b"]=42; for(i in a) print i, a[i]}'
z 1
x 12
b 42
```

split

The `split` function provides the same features as the record splitting done using `FS`. This is helpful when you need the results as an array for some reason, for example to use array sorting features. Or, when you need to split some string in addition to record splitting. `split` accepts four arguments, with last two being optional.

- First argument is the string to be split
- Second argument is the array variable to save results
- Third argument is the separator, whose default is `FS`

The return value of `split` function is number of fields, similar to `NF` variable. The array gets indexed starting from `1` for first element, `2` for second element and so on. If the array already had some value, it gets overwritten with the new value.

```
$ # same as: awk '{print $2}'
$ printf '     one \t two\t\t\tthree  ' | awk '{split($0, a); print a[2]}'
two

$ # example with both FS and split in action
$ s='Joe,1996-10-25,64,78'
$ echo "$s" | awk -F, '{split($2, d, "-"); print $1 " was born in " d[1]}'
Joe was born in 1996

$ # single row to multiple rows based on splitting last field
$ s='air,water,12:42:3'
$ echo "$s" | awk -F, '{n=split($NF, a, ":");
                        for(i=1; i<=n; i++) print $1,$2,a[i]}'
air water 12
air water 42
air water 3
```

Similar to `FS`, you can use regular expression as a separator.

```
$ s='Sample123string42with777numbers'
$ echo "$s" | awk '{split($0, s, /[0-9]+/); print s[2], s[4]}'
```

```
string numbers
```

The fourth argument provides a feature not present with record splitting. It allows you to save the portions matched by the separator in an array.

```
$ # note that it is i<n not i<=n
$ s='Sample123string42with777numbers'
$ echo "$s" | awk '{n=split($0, s, /[0-9]+/, seps);
                  for(i=1; i<n; i++) print seps[i]}'
123
42
777
```

Here's an example where `split` is merely used to initialize an array based on empty separator. Unlike `$N` syntax where an expression resulting in floating-point number is acceptable, array index has to be an integer. Hence, `int` function is used to convert floating-point result to integer in the example below.

```
$ cat marks.txt
Dept    Name    Marks
ECE     Raj     53
ECE     Joel    72
EEE     Moi     68
CSE     Surya   81
EEE     Tia     59
ECE     Om      92
CSE     Amy     67

$ # adds a new grade column based on marks in 3rd column
$ awk 'BEGIN{OFS="\t"; split("DCBAS", g, //)}
       {$(NF+1) = NR==1 ? "Grade" : g[int($NF/10)-4]} 1' marks.txt
Dept    Name    Marks   Grade
ECE     Raj     53      D
ECE     Joel    72      B
EEE     Moi     68      C
CSE     Surya   81      A
EEE     Tia     59      D
ECE     Om      92      S
CSE     Amy     67      C
```

patsplit

The `patsplit` function will give you the features provided by `FPAT`. The argument order and optional arguments is same as the `split` function, with `FPAT` as the default separator. The return value is number of fields obtained from the split.

```
$ s='eagle,"fox,42",bee,frog'

$ echo "$s" | awk '{patsplit($0, a, /"[^"]*"|[^,]*/); print a[2]}'
"fox,42"
```

substr

The `substr` function allows to extract specified number of characters from given string based on indexing. The argument order is:

- First argument is the input string
- Second argument is starting position
- Third argument is number of characters to extract

The index starts from `1`. If the third argument is not specified, by default all characters until the end of string input is extracted. If the second argument is greater than length of the string or if third argument is less than or equal to `0` then empty string is returned. Second argument will use `1` if a number less than one is specified.

```
$ echo 'abcdefghij' | awk '{print substr($0, 1, 5)}'
abcde
$ echo 'abcdefghij' | awk '{print substr($0, 4, 3)}'
def

$ echo 'abcdefghij' | awk '{print substr($0, 6)}'
fghij

$ echo 'abcdefghij' | awk -v OFS=: '{print substr($0, 2, 3), substr($0, 6, 3)}'
bcd:fgh
```

If only a few characters are needed from input record, can also use empty `FS`.

```
$ echo 'abcdefghij' | awk -v FS= '{print $3}'
c
$ echo 'abcdefghij' | awk -v FS= '{print $3, $5}'
c e
```

match

The `match` function is useful to extract portion of an input string matched by a regexp. There are two ways to get the matched portion:

- by using `substr` function along with special variables `RSTART` and `RLENGTH`
- by passing a third argument to `match` so that the results are available from an array

The first argument to `match` is the input string and second is the regexp. If the match fails, then `RSTART` gets `0` and `RLENGTH` gets `-1`. Return value is same as `RSTART`.

```
$ s='051 035 154 12 26 98234'

$ # using substr and RSTART/RLENGTH
$ echo "$s" | awk 'match($0, /[0-9]{4,}/){print substr($0, RSTART, RLENGTH)}'
98234

$ # using array, note that index 0 is used here, not 1
$ echo "$s" | awk 'match($0, /0*[1-9][0-9]{2,}/, m){print m[0]}'
154
```

Both the above examples can also be easily solved using `FPAT` or `patsplit`. `match` has an advantage when it comes to getting portions matched only within capture groups. The first element of array will still have the entire match. Second element will contain portion matched by first group, third element will contain portion matched by second group and so on.

```
$ # entire matched portion
$ echo 'foo=42, baz=314' | awk 'match($0, /baz=([0-9]+)/, m){print m[0]}'
baz=314

$ # matched portion of first capture group
$ echo 'foo=42, baz=314' | awk 'match($0, /baz=([0-9]+)/, m){print m[1]}'
314
```

If you need to get matching portions for all the matches instead of just the first match, you can use a loop and adjust the input string every iteration.

```
$ # extract numbers only if it is followed by a comma
$ s='42 foo-5, baz3; x-83, y-20: f12'
$ echo "$s" | awk '{ while( match($0, /([0-9]+),/, m) ){print m[1];
                  $0=substr($0, RSTART+RLENGTH)} }'
5
83
```

index

The `index` function is useful when you need to match a string literally in the given input string. This is similar to `grep -F` functionality of matching fixed strings. The first argument to this function is the input string and the second is the string to be matched literally. The return value is the index of matching location and `0` if there is no match.

```
$ cat eqns.txt
a=b,a-b=c,c*d
a+b,pi=3.14,5e12
i*(t+9-g)/8,4-a+b

$ # no output because metacharacters aren't escaped
$ awk '/i*(t+9-g)/' eqns.txt
$ # same as: grep -F 'i*(t+9-g)' eqns.txt
$ awk 'index($0, "i*(t+9-g)")' eqns.txt
i*(t+9-g)/8,4-a+b

$ # check only last field
$ awk -F, 'index($NF, "a+b")' eqns.txt
i*(t+9-g)/8,4-a+b
$ # index not needed if entire field/line is being compared
$ awk -F, '$1=="a+b"' eqns.txt
a+b,pi=3.14,5e12
```

The return value is also useful to ensure match is found at specific positions only. For example start or end of input string.

```
$ # start of line
$ awk 'index($0, "a+b")==1' eqns.txt
a+b,pi=3.14,5e12
$ # end of line
$ awk -v s="a+b" 'index($0, s)==length()-length(s)+1' eqns.txt
i*(t+9-g)/8,4-a+b
```

Recall that `-v` option gets parsed by `awk` 's string processing rules. So, if you need to pass a literal string without falling in backslash hell, use `ENVIRON` instead of `-v` option.

```
$ echo 'a\b\c\d' | awk -v s='a\b' 'index($0, s)'
$ echo 'a\b\c\d' | awk -v s='a\\b' 'index($0, s)'
a\b\c\d

$ echo 'a\b\c\d' | s='a\b' awk 'index($0, ENVIRON["s"])'
a\b\c\d
```

system

External commands can be issued using the `system` function. Any output generated by the external command would be as usual on `stdout` unless redirected while calling the command.

```
$ awk 'BEGIN{system("echo Hello World")}'
Hello World

$ wc table.txt
 3 15 79 table.txt
$ awk 'BEGIN{system("wc table.txt")}'
 3 15 79 table.txt

$ awk 'BEGIN{system("seq 10 | paste -sd, > out.txt")}'
$ cat out.txt
1,2,3,4,5,6,7,8,9,10

$ cat f2.txt
I bought two balls and 3 bats
$ echo 'f1,f2,f3' | awk -F, '{system("cat " $2 ".txt")}'
I bought two balls and 3 bats
```

Return value of `system` depends on `exit` status of the executed command. See gawk manual: Input/Output Functions for details.

```
$ ls xyz.txt
ls: cannot access 'xyz.txt': No such file or directory
$ echo $?
2

$ awk 'BEGIN{s=system("ls xyz.txt"); print "Exit status: " s}'
ls: cannot access 'xyz.txt': No such file or directory
Exit status: 2
```

printf and sprintf

The `printf` function is useful over `print` function when you need to format the data before printing. Another difference is that `OFS` and `ORS` do not affect the `printf` function. The features are similar to those found in `C` programming language and the shell built-in command.

```
$ # default behavior with print function
$ # number of digits after decimal point varies based on length
$ awk 'BEGIN{sum = 3.1428 + 10; print sum}'
13.1428
$ awk 'BEGIN{sum = 3.1428 + 100; print sum}'
103.143

$ # use printf function for finer control
$ # note the use of \n as ORS isn't appended unlike print
$ awk 'BEGIN{sum = 3.1428 + 10; printf "%f\n", sum}'
13.142800
$ awk 'BEGIN{sum = 3.1428 + 10; printf "%.3f\n", sum}'
13.143
```

Here's some more formatting options for floating-point numbers.

```
$ # total length is 10, filled with space if needed
$ # [ and ] are used here for visualization purposes
$ awk 'BEGIN{pi = 3.14159; printf "[%10.3f]\n", pi}'
[     3.142]
$ awk 'BEGIN{pi = 3.14159; printf "[%-10.3f]\n", pi}'
[3.142     ]

$ # zero filled
$ awk 'BEGIN{pi = 3.14159; printf "%010.3f\n", pi}'
000003.142

$ # scientific notation
$ awk 'BEGIN{pi = 3.14159; printf "%e\n", pi}'
3.141590e+00
```

Here's some formatting options for integers.

```
$ # note that there is no rounding
$ awk 'BEGIN{printf "%d\n", 1.99}'
1

$ # ensure there's always a sign prefixed to integer
$ awk 'BEGIN{printf "%+d\n", 100}'
+100
$ awk 'BEGIN{printf "%+d\n", -100}'
-100
```

Here's some formatting options for strings.

```
$ # prefix remaining width with spaces
$ awk 'BEGIN{printf "|%10s|\n", "mango"}'
|     mango|

$ # suffix remaining width with spaces
$ awk 'BEGIN{printf "|%-10s|\n", "mango"}'
|mango     |

$ # truncate
$ awk '{printf "%.4s\n", $0}' table.txt
brow
blue
yell
```

You can also refer to an argument using `N$` format, where `N` is the positional number of argument. One advantage with this method is that you can reuse an argument any number of times. You cannot mix this format with the normal way.

```
$ awk 'BEGIN{printf "%1$d + %2$d * %1$d = %3$d\n", 3, 4, 15}'
3 + 4 * 3 = 15

$ # remove # if you do not need the prefix
$ awk 'BEGIN{printf "hex=%1$#x\noct=%1$#o\ndec=%1$d\n", 15}'
hex=0xf
oct=017
dec=15
```

You can pass variables by specifying a `*` instead of a number in the formatting string.

```
$ # same as: awk 'BEGIN{pi = 3.14159; printf "%010.3f\n",  pi}'
$ awk 'BEGIN{d=10; p=3; pi = 3.14159; printf "%0*.*f\n", d, p, pi}'
000003.142
```

Passing a variable directly to `printf` without using a format specifier can result in error depending upon the contents of the variable.

```
$ awk 'BEGIN{s="solve: 5 % x = 1"; printf s}'
awk: cmd. line:1: fatal: not enough arguments to satisfy format string
    `solve: 5 % x = 1'
               ^ ran out for this one
```

So, as a good practice, always use variables with appropriate format instead of passing it directly to `printf`.

```
$ awk 'BEGIN{s="solve: 5 % x = 1"; printf "%s\n", s}'
solve: 5 % x = 1
```

If `%` has to be used literally inside the format specifier, use `%%`. This is similar to using `\\` in regexp to represent `\` literally.

```
$ awk 'BEGIN{printf "n%%d gives the remainder\n"}'
n%d gives the remainder
```

To save the results of the formatting in a variable instead of printing, use `sprintf` function.

Unlike `printf` , parentheses are always required to use `sprintf` function.

```
$ awk 'BEGIN{pi = 3.14159; s = sprintf("%010.3f", pi); print s}'
000003.142
```

See gawk manual: printf for complete list of formatting options and other details.

Redirecting print output

The results from `print` and `printf` functions can be redirected to a shell command or a file instead of `stdout` . There's nothing special about it, you could have done it normally on `awk` command as well. The use case arises when you need multiple redirections within the same `awk` command. Here's some examples of redirecting to multiple files.

```
$ seq 6 | awk 'NR%2{print > "odd.txt"; next} {print > "even.txt"}'
$ cat odd.txt
1
3
5
$ cat even.txt
2
4
6

$ # dynamically creating filenames
$ awk -v OFS='\t' 'NR>1{print $2, $3 > $1".txt"}' marks.txt
$ # output for one of the departments
$ cat ECE.txt
Raj     53
Joel    72
Om      92
```

Note that the use of `>` doesn't mean that the file will get overwritten everytime. That happens only once if the file already existed prior to executing the `awk` command. Use `>>` if you wish to append to already existing files.

As seen in above examples, the file names are passed as string expressions. To redirect to a shell command, again you need to pass a string expression after `|` pipe symbol. Here's an example.

```
$ awk '{print $2 | "paste -sd,"}' table.txt
bread,cake,banana
```

And here's some examples of multiple redirections.

```
$ awk '{print $2 | "sort | paste -sd,"}' table.txt
banana,bread,cake

$ # sort the output before writing to files
$ awk -v OFS='\t' 'NR>1{print $2, $3 | "sort > "$1".txt"}' marks.txt
$ # output for one of the departments
```

```
$ cat ECE.txt
Joel    72
Om      92
Raj     53
```

 See gawk manual: Redirecting Output of print and printf for more details and operators on redirections. And see gawk manual: Closing Input and Output Redirections if you have too many redirections.

Summary

This chapter covered some of the built-in functions provided by `awk` . Do check the manual for more of them, for example math and time related functions.

Next chapter will cover features related to processing multiple files passed as input to `awk` .

Multiple file input

You have already seen control structures like `BEGIN`, `END` and `next`. This chapter will discuss control structures that are useful to make decisions around each file when there are multiple files passed as input.

BEGINFILE, ENDFILE and FILENAME

- `BEGINFILE` — this block gets executed before start of each input file
- `ENDFILE` — this block gets executed after processing each input file
- `FILENAME` — special variable having file name of current input file

```
$ awk 'BEGINFILE{print "--- " FILENAME " ---"} 1' greeting.txt table.txt
--- greeting.txt ---
Hi there
Have a nice day
Good bye
--- table.txt ---
brown bread mat hair 42
blue cake mug shirt -7
yellow banana window shoes 3.14

$ # same as: tail -q -n1 greeting.txt table.txt
$ awk 'ENDFILE{print $0}' greeting.txt table.txt
Good bye
yellow banana window shoes 3.14
```

nextfile

`nextfile` will skip remaining records from current file being processed and move on to next file.

```
$ # print filename if it contains '3' anywhere in the file
$ # same as: grep -l '3' f[1-3].txt greeting.txt
$ awk '/3/{print FILENAME; nextfile}' f[1-3].txt greeting.txt
f1.txt
f2.txt

$ # print filename if it contains both 'o' and 'at' anywhere in the file
$ awk 'BEGINFILE{m1=m2=0} /o/{m1=1} /at/{m2=1}
       m1 && m2{print FILENAME; nextfile}' f[1-3].txt greeting.txt
f2.txt
f3.txt
```

ARGC and ARGV

The `ARGC` special variable contains total number of arguments passed to the `awk` command, including `awk` itself as an argument. The `ARGV` special array contains the arguments themselves.

```
$ # note that index starts with '0' here
$ awk 'BEGIN{for(i=0; i<ARGC; i++) print ARGV[i]}' f[1-3].txt greeting.txt
awk
f1.txt
f2.txt
f3.txt
greeting.txt
```

Similar to manipulating `NF` and modifying `$N` field contents, you can change the values of `ARGC` and `ARGV` to control how the arguments should be processed.

However, not all arguments are necessarily filenames. `awk` allows assigning variable values without `-v` option if it is done in the place where you usually provide file arguments. For example:

```
$ awk 'BEGIN{for(i=0; i<ARGC; i++) print ARGV[i]}' table.txt n=5 greeting.txt
awk
table.txt
n=5
greeting.txt
```

In the above example, the variable `n` will get a value of `5` after `awk` has finished processing `table.txt` file. Here's an example where `FS` is changed between two files.

```
$ cat table.txt
brown bread mat hair 42
blue cake mug shirt -7
yellow banana window shoes 3.14
$ cat books.csv
Harry Potter,Mistborn,To Kill a Mocking Bird
Matilda,Castle Hangnail,Jane Eyre

$ # for table.txt, FS will be default value
$ # for books.csv, FS will be comma character
$ # OFS is comma for both files
$ awk -v OFS=, 'NF=2' table.txt FS=, books.csv
brown,bread
blue,cake
yellow,banana
Harry Potter,Mistborn
Matilda,Castle Hangnail
```

Summary

This chapter introduced few more special blocks and variables are that handy for processing multiple file inputs. These will show up in examples in coming chapters as well.

Next chapter will discuss use cases where you need to take decisions based on multiple input records.

Processing multiple records

Often, you need to consider multiple lines at a time to make a decision, such as the paragraph mode examples seen earlier. Sometimes, you need to match a particular record and then get records surrounding the matched record. The `condX{actionX}` shortcut makes it easy to code state machines concisely, which is useful to solve such multiple record use cases. See softwareengineering: FSM examples if you are not familiar with state machines.

Processing consecutive records

You might need to define a condition that should satisfy something for one record and something else for the very next record. `awk` does provide a feature to get next record, but that could get complicated (see getline section). Instead, you can simply save each record in a variable and then create the required conditional expression. The default behavior of uninitialized variable to act as `0` in numerical context and empty in string context plays a role too.

```
$ # match and print two consecutive records
$ # first record should contain 'as' and second record should contain 'not'
$ awk 'p ~ /as/ && /not/{print p ORS $0} {p=$0}' programming_quotes.txt
Therefore, if you write the code as cleverly as possible, you are,
by definition, not smart enough to debug it by Brian W. Kernighan

$ # same filtering as above, but print only the first record
$ awk 'p ~ /as/ && /not/{print p} {p=$0}' programming_quotes.txt
Therefore, if you write the code as cleverly as possible, you are,

$ # same filtering as above, but print only the second record
$ awk 'p ~ /as/ && /not/; {p=$0}' programming_quotes.txt
by definition, not smart enough to debug it by Brian W. Kernighan
```

Context matching

Sometimes you want not just the matching records, but the records relative to the matches as well. For example, it could be to see the comments at start of a function block that was matched while searching a program file. Or, it could be to see extended information from a log file while searching for a particular error message.

Consider this sample input file:

```
$ cat context.txt
blue
    toy
    flower
    sand stone
light blue
    flower
    sky
    water
```

```
language
    english
    hindi
    spanish
    tamil
programming language
    python
    kotlin
    ruby
```

Here's an example that emulates `grep --no-group-separator -A<n>` functionality. The `n && n--` trick used in the example works like this:

- If initially `n=2` , then we get
  - `2 && 2` --> evaluates to `true` and `n` becomes `1`
  - `1 && 1` --> evaluates to `true` and `n` becomes `0`
  - `0 &&` --> evaluates to `false` and `n` doesn't change
- Note that when conditionals are connected with logical `&&` , the right expression will not be executed at all if the left one turns out to be `false` because the overall result will always be `false` . Same is the case if left expression evaluates to `true` with logical `||` operator. Such logical operators are also known as **short-circuit** operators. Thus, in the above case, `n--` won't be executed when `n` is `0` on the left hand side. This prevents `n` going negative and `n && n--` will never become `true` unless `n` is assigned again.

```
$ # same as: grep --no-group-separator -A1 'blue'
$ # print matching line as well as the one that follows it
$ awk '/blue/{n=2} n && n--' context.txt
blue
    toy
light blue
    flower

$ # overlapping example, n gets re-assigned before reaching 0
$ awk '/toy|flower/{n=2} n && n--{print NR, $0}' context.txt
2     toy
3     flower
4     sand stone
6     flower
7     sky

$ # doesn't allow overlapping cases to re-assign the counter
$ awk '!n && /toy|flower/{n=2} n && n--{print NR, $0}' context.txt
2     toy
3     flower
6     flower
7     sky
```

Once you've understood the above examples, the rest of the examples in this section should be easier to comprehend. They are all variations of the logic used above and re-arranged to

solve the use case being discussed.

Print `n` records after match. This is similar to previous case, except that the matching record isn't printed.

```
$ # print 1 line after matching line
$ # for overlapping cases, n gets re-assigned before reaching 0
$ awk 'n && n--; /language/{n=1}' context.txt
    english
    python

$ # print 2 lines after matching line
$ # doesn't allow overlapping cases to re-assign the counter
$ awk '!n && /toy|flower/{n=2; next} n && n--' context.txt
    flower
    sand stone
    sky
    water
```

Here's how to print `n` th record after the matching record.

```
$ # print only the 2nd line found after matching line
$ # the array saves matching result for each record
$ # doesn't rely on a counter, thus works for overlapping cases
$ awk -v n=2 'a[NR-n]; /toy|flower/{a[NR]=1}' context.txt
    sand stone
light blue
    water

$ # print only the 3rd line found after matching line
$ # n && !--n will be true only when --n yields 0
$ # overlapping cases won't work as n gets re-assigned before going to 0
$ awk 'n && !--n; /language/{n=3}' context.txt
    spanish
    ruby
```

Print `n` records before match. Printing the matching record as well is left as an exercise. Since the file is being read in forward direction, and the problem statement is to print something before the matching record, overlapping situation like the previous examples doesn't occur.

```
$ # i>0 is used because NR starts from 1
$ awk -v n=2 '/toy|flower/{for(i=NR-n; i<NR; i++) if(i>0) print a[i]}
              {a[NR]=$0}' context.txt
blue
blue
    toy
    sand stone
light blue
```

Print `n` th record before the matching record.

```
$ # if the count is small enough, you can save them in variables
$ # this one prints 2nd line before the matching line
$ # NR>2 is needed as first 2 records shouldn't be considered for a match
$ awk 'NR>2 && /toy|flower/{print p2} {p2=p1; p1=$0}' context.txt
blue
    sand stone

$ # else, use an array to save previous records
$ awk -v n=4 'NR>n && /age/{print a[NR-n]} {a[NR]=$0}' context.txt
light blue
    english
```

Records bounded by distinct markers

This section will cover cases where the input file will always contain the same number of starting and ending patterns and arranged in alternating fashion. For example, there cannot be two starting patterns appearing without an ending pattern between them and vice versa. Zero or more records of text can appear inside such groups as well as in between the groups.

The sample file shown below will be used to illustrate examples in this section. For simplicity, assume that the starting pattern is marked by `start` and the ending pattern by `end`. They have also been given group numbers to make it easier to visualize the transformation between input and output for the commands discussed in this section.

```
$ cat uniform.txt
mango
icecream
--start 1--
1234
6789
**end 1**
how are you
have a nice day
--start 2--
a
b
c
**end 2**
par,far,mar,tar
```

Case 1: Processing all the groups of records based on the distinct markers, including the records matched by markers themselves. For simplicity, the below command will just print all such records.

```
$ awk '/start/{f=1} f; /end/{f=0}' uniform.txt
--start 1--
1234
6789
**end 1**
--start 2--
```

```
a
b
c
**end 2**
```


Similar to `sed -n '/start/,/end/p'` you can also use `awk '/start/,/end/'` but the state machine format is more suitable to change for various cases to follow.

Case 2: Processing all the groups of records but excluding the records matched by markers themselves.

```
$ awk '/end/{f=0} f{print "*", $0} /start/{f=1}' uniform.txt
* 1234
* 6789
* a
* b
* c
```

Case 3-4: Processing all the groups of records but excluding either of the markers.

```
$ awk '/start/{f=1} /end/{f=0} f' uniform.txt
--start 1--
1234
6789
--start 2--
a
b
c

$ awk 'f; /start/{f=1} /end/{f=0}' uniform.txt
1234
6789
**end 1**
a
b
c
**end 2**
```

The next four cases are obtained by just using `!f` instead of `f` from the cases shown above.

Case 5: Processing all input records except the groups of records bound by the markers.

```
$ awk '/start/{f=1} !f{print $0 "."} /end/{f=0}' uniform.txt
mango.
icecream.
how are you.
have a nice day.
par,far,mar,tar.
```

Case 6 Processing all input records except the groups of records between the markers.

```
$ awk '/end/{f=0} !f; /start/{f=1}' uniform.txt
mango
```

```
icecream
--start 1--
**end 1**
how are you
have a nice day
--start 2--
**end 2**
par,far,mar,tar
```

Case 7-8: Similar to case 6, but include only one of the markers.

```
$ awk '!f; /start/{f=1} /end/{f=0}' uniform.txt
mango
icecream
--start 1--
how are you
have a nice day
--start 2--
par,far,mar,tar

$ awk '/start/{f=1} /end/{f=0} !f' uniform.txt
mango
icecream
**end 1**
how are you
have a nice day
**end 2**
par,far,mar,tar
```

Specific blocks

Instead of working with all the groups (or blocks) bound by the markers, this section will discuss how to choose blocks based on additional criteria.

Here's how you can process only the first matching block.

```
$ awk '/start/{f=1} f; /end/{exit}' uniform.txt
--start 1--
1234
6789
**end 1**

$ # use other tricks discussed in previous section as needed
$ awk '/end/{exit} f; /start/{f=1}' uniform.txt
1234
6789
```

Getting last block alone involves lot more work, unless you happen to know how many blocks are present in the input file.

```
$ # reverse input linewise, change the order of comparison, reverse again
$ # can't be used if RS has to be something other than newline
$ tac uniform.txt | awk '/end/{f=1} f; /start/{exit}' | tac
--start 2--
a
b
c
**end 2**

$ # or, save the blocks in a buffer and print the last one alone
$ awk '/start/{f=1; b=$0; next} f{b=b ORS $0} /end/{f=0}
       END{print b}' uniform.txt
--start 2--
a
b
c
**end 2**
```

Only the `n` th block.

```
$ # can also use: awk -v n=2 '/4/{c++} c==n{print; if(/6/) exit}'
$ seq 30 | awk -v n=2 '/4/{c++} c==n; /6/ && c==n{exit}'
14
15
16
```

All blocks greater than `n` th block.

```
$ seq 30 | awk -v n=1 '/4/{f=1; c++} f && c>n; /6/{f=0}'
14
15
16
24
25
26
```

Excluding `n` th block.

```
$ seq 30 | awk -v n=2 '/4/{f=1; c++} f && c!=n; /6/{f=0}'
4
5
6
24
25
26
```

All blocks, only if the records between the markers match an additional condition.

```
$ # additional condition here is a line starting with '1'
$ seq 30 | awk '/4/{f=1; buf=$0; m=0; next}
                f{buf=buf ORS $0}
                /6/{f=0; if(buf && m) print buf; buf=""}
                /^1/{m=1}'
```

```
14
15
16
```

Broken blocks

Sometimes, you can have markers in random order and mixed in different ways. In such cases, to work with blocks without any other marker present in between them, the buffer approach comes in handy again.

```
$ cat broken.txt
qqqqqqqqqqqqqqq
error 1
hi
error 2
1234
6789
state 1
bye
state 2
error 3
xyz
error 4
abcd
state 3
zzzzzzzzzzzzzzz

$ awk '/error/{f=1; buf=$0; next}
       f{buf=buf ORS $0}
       /state/{f=0; if(buf) print buf; buf=""}' broken.txt
error 2
1234
6789
state 1
error 4
abcd
state 3
```

Summary

This chapter covered various examples of working with multiple records. State machines play an important role in deriving solutions for such cases. Knowing various corner cases is also crucial, otherwise a solution that works for one input may fail for others.

Next chapter will discuss use cases where you need to process a file input based on contents of another file.

Two file processing

This chapter focuses on solving problems which depend upon contents of two files. These are usually based on comparing records and fields. Sometimes, record number plays a role too. You'll also learn about the `getline` built-in function.

Comparing records

Consider the following input files which will be compared line wise to get common lines and unique lines.

```
$ cat color_list1.txt
teal
light blue
green
yellow

$ cat color_list2.txt
light blue
black
dark green
yellow
```

The *key* features used in the solution below:

- For two files as input, `NR==FNR` will be `true` only when the first file is being processed
- `next` will skip rest of code and fetch next record
- `a[$0]` by itself is a valid statement. It will create an uninitialized element in array `a` with `$0` as the key (assuming the key doesn't exist yet)
- `$0 in a` checks if the given string (`$0` here) exists as a key in array `a`

```
$ # common lines
$ # same as: grep -Fxf color_list1.txt color_list2.txt
$ awk 'NR==FNR{a[$0]; next} $0 in a' color_list1.txt color_list2.txt
light blue
yellow

$ # lines from color_list2.txt not present in color_list1.txt
$ # same as: grep -vFxf color_list1.txt color_list2.txt
$ awk 'NR==FNR{a[$0]; next} !($0 in a)' color_list1.txt color_list2.txt
black
dark green

$ # reversing the order of input files gives
$ # lines from color_list1.txt not present in color_list2.txt
$ awk 'NR==FNR{a[$0]; next} !($0 in a)' color_list2.txt color_list1.txt
teal
green
```

Comparing fields

In the previous section, you saw how to compare whole contents of records between two files. This section will focus on comparing only specific field(s). The below sample file will be one of the two file inputs for examples in this section.

```
$ cat marks.txt
Dept    Name    Marks
ECE     Raj     53
ECE     Joel    72
EEE     Moi     68
CSE     Surya   81
EEE     Tia     59
ECE     Om      92
CSE     Amy     67
```

To start with, here's a single field comparison. The problem statement is to fetch all records from `marks.txt` if the first field matches any of the departments listed in `dept.txt` file.

```
$ cat dept.txt
CSE
ECE

$ # note that dept.txt is used to build the array keys first
$ awk 'NR==FNR{a[$1]; next} $1 in a' dept.txt marks.txt
ECE     Raj     53
ECE     Joel    72
CSE     Surya   81
ECE     Om      92
CSE     Amy     67

$ # if header is needed as well
$ awk 'NR==FNR{a[$1]; next} FNR==1 || $1 in a' dept.txt marks.txt
Dept    Name    Marks
ECE     Raj     53
ECE     Joel    72
CSE     Surya   81
ECE     Om      92
CSE     Amy     67
```

With multiple field comparison, constructing the key might become important for some cases. If you simply concatenate field values, it may lead to false matches. For example, field values `abc` and `123` will wrongly match `ab` and `c123`. To avoid this, you may introduce some string between the field values, say `"_"` (if you know the field themselves cannot have this character) or `FS` (safer option). You could also allow `awk` to bail you out. If you use `,` symbol (not `","` as a string) between field values, the value of special variable `SUBSEP` is inserted. `SUBSEP` has a default value of the non-printing character `\034` which is usually not used as part of text files.

```
$ cat dept_name.txt
EEE Moi
```

```
CSE Amy
ECE Raj

$ # uses SUBSEP as separator between field values to construct the key
$ # note the use of parentheses for key testing
$ awk 'NR==FNR{a[$1,$2]; next} ($1,$2) in a' dept_name.txt marks.txt
ECE     Raj     53
EEE     Moi     68
CSE     Amy     67
```

In this example, one of the field is used for numerical comparison.

```
$ cat dept_mark.txt
ECE 70
EEE 65
CSE 80

$ # match Dept and minimum marks specified in dept_mark.txt
$ awk 'NR==FNR{d[$1]=$2; next}
       $1 in d && $3 >= d[$1]' dept_mark.txt marks.txt
ECE     Joel    72
EEE     Moi     68
CSE     Surya   81
ECE     Om      92
```

Here's an example of adding a new field.

```
$ cat role.txt
Raj class_rep
Amy sports_rep
Tia placement_rep

$ awk -v OFS='\t' 'NR==FNR{r[$1]=$2; next}
         {$(NF+1) = FNR==1 ? "Role" : r[$2]} 1' role.txt marks.txt
Dept    Name    Marks   Role
ECE     Raj     53      class_rep
ECE     Joel    72
EEE     Moi     68
CSE     Surya   81
EEE     Tia     59      placement_rep
ECE     Om      92
CSE     Amy     67      sports_rep
```

getline

As the name indicates, `getline` function allows you to read a line from a file on demand. This is most useful when you need something based on line number. The following example shows how you can replace `m` th line from a file with `n` th line from another file. There are many syntax variations with `getline`, here the line read is saved in a variable.

```
$ # return value handling is not shown here, but should be done ideally
$ awk -v m=3 -v n=2 'BEGIN{while(n-- > 0) getline s < "greeting.txt"}
                    FNR==m{$0=s} 1' table.txt
brown bread mat hair 42
blue cake mug shirt -7
Have a nice day
```

Here's an example where two files are processed simultaneously. In this case, the return value of `getline` is also used. It will be `1` if line was read successfully, `0` if there's no more input to be read as end of file has already been reached and `-1` if something went wrong. The `ERRNO` special variable will have details regarding the issues.

```
$ # print line from greeting.txt if last column of corresponding line
$ # from table.txt is +ve number
$ awk -v file='table.txt' '(getline line < file)==1{n=split(line, a);
                          if(a[n]>0) print}' greeting.txt
Hi there
Good bye
```

If a file is passed as argument to `awk` command and cannot be opened, you get an error. For example:

```
$ awk '{print $2}' xyz.txt
awk: fatal: cannot open file `xyz.txt' for reading (No such file or directory)
```

It is recommended to always check for return value when using `getline` or perhaps use techniques from previous sections to avoid `getline` altogether.

```
$ # xyz.txt doesn't exist, but output doesn't show something went wrong
$ awk '{getline line < "xyz.txt"; print $NF, line}' table.txt
42
-7
3.14

$ awk -v file='xyz.txt' '{ e=(getline line < file);
                           if(e<0){print file ": " ERRNO; exit}
                           print $NF, line }' table.txt
xyz.txt: No such file or directory
```


See gawk manual: getline for details, especially about corner cases and errors. See also awk.freeshell: getline caveats.

Summary

This chapter discussed a few cases where you need to compare contents of two files. The `NR==FNR` trick is handy for such cases. The `getline` function is helpful for line number based comparisons.

Next chapter will discuss how to handle duplicate contents.

Dealing with duplicates

Often, you need to eliminate duplicates from an input file. This could be based on entire line content or based on certain fields. These are typically solved with `sort` and `uniq` commands. Advantage with `awk` include regexp based field and record separators, input doesn't have to be sorted, and in general more flexibility because it is a programming language.

Whole line duplicates

`awk '!a[$0]++'` is one of the most famous `awk` one-liners. It eliminates line based duplicates while retaining input order. The following example shows it in action along with an illustration of how the logic works.

```
$ cat purchases.txt
coffee
tea
washing powder
coffee
toothpaste
tea
soap
tea

$ awk '{print +a[$0] "\t" $0; a[$0]++}' purchases.txt
0       coffee
0       tea
0       washing powder
1       coffee
0       toothpaste
1       tea
0       soap
2       tea

$ # only those entries with zero in first column will be retained
$ awk '!a[$0]++' purchases.txt
coffee
tea
washing powder
toothpaste
soap
```

Column wise duplicates

Removing field based duplicates is simple for single field comparison. Just change `$0` to the required field number after setting the appropriate field separator.

```
$ cat duplicates.txt
brown,toy,bread,42
```

```
dark red,ruby,rose,111
blue,ruby,water,333
dark red,sky,rose,555
yellow,toy,flower,333
white,sky,bread,111
light red,purse,rose,333

$ # based on last field
$ awk -F, '!seen[$NF]++' duplicates.txt
brown,toy,bread,42
dark red,ruby,rose,111
blue,ruby,water,333
dark red,sky,rose,555
```

For multiple fields comparison, separate the fields with `,` so that `SUBSEP` is used to combine the field value to generate the key. As mentioned before, `SUBSEP` has a default value of `\034` non-printing character, which is typically not used in text files.

```
$ # based on first and third field
$ awk -F, '!seen[$1,$3]++' duplicates.txt
brown,toy,bread,42
dark red,ruby,rose,111
blue,ruby,water,333
yellow,toy,flower,333
white,sky,bread,111
light red,purse,rose,333
```

Duplicate count

In this section, how many times a duplicate record is found plays a role in determining the output.

First up, printing only a specific numbered duplicate.

```
$ # print only the second occurrence of duplicates based on 2nd field
$ awk -F, '++seen[$2]==2' duplicates.txt
blue,ruby,water,333
yellow,toy,flower,333
white,sky,bread,111

$ # print only the third occurrence of duplicates based on last field
$ awk -F, '++seen[$NF]==3' duplicates.txt
light red,purse,rose,333
```

Next, printing only the last copy of duplicate. Since the count isn't known, the `tac` command comes in handy again.

```
$ # reverse the input line-wise, retain first copy and then reverse again
$ tac duplicates.txt | awk -F, '!seen[$NF]++' | tac
brown,toy,bread,42
dark red,sky,rose,555
```

```
white,sky,bread,111
light red,purse,rose,333
```

To get all the records based on a duplicate count, you can pass the input file twice. Then use the two file processing trick to make decisions.

```
$ # all duplicates based on last column
$ awk -F, 'NR==FNR{a[$NF]++; next} a[$NF]>1' duplicates.txt duplicates.txt
dark red,ruby,rose,111
blue,ruby,water,333
yellow,toy,flower,333
white,sky,bread,111
light red,purse,rose,333

$ # all duplicates based on last column, minimum 3 duplicates
$ awk -F, 'NR==FNR{a[$NF]++; next} a[$NF]>2' duplicates.txt duplicates.txt
blue,ruby,water,333
yellow,toy,flower,333
light red,purse,rose,333

$ # only unique lines based on 3rd column
$ awk -F, 'NR==FNR{a[$3]++; next} a[$3]==1' duplicates.txt duplicates.txt
blue,ruby,water,333
yellow,toy,flower,333
```

Summary

This chapter showed how to work with duplicate contents, both record and field based. If you don't need regexp based separators and if your input is too big to handle, then specialized command line tools `sort` and `uniq` will be better suited compared to `awk` .

Next chapter will show how to write `awk` scripts instead of the usual one-liners.

awk scripts

-f option

The `-f` command line option allows you to pass the `awk` code via file instead of writing it all on the command line. Here's a one-liner seen earlier that's been converted to a multiline script. Note that `;` is no longer necessary to separate the commands, newline will do that too.

```
$ cat buf.awk
/error/{
    f = 1
    buf = $0
    next
}

f{
    buf = buf ORS $0
}

/state/{
    f = 0
    if(buf)
        print buf
    buf = ""
}

$ awk -f buf.awk broken.txt
error 2
1234
6789
state 1
error 4
abcd
state 3
```

Another advantage is that single quotes can be freely used.

```
$ echo 'cue us on this example' | awk -v q="'" '{gsub(/\w+/, q "&" q)} 1'
'cue' 'us' 'on' 'this' 'example'

$ cat quotes.awk
{
    gsub(/\w+/, "'&'")
}

1

$ echo 'cue us on this example' | awk -f quotes.awk
'cue' 'us' 'on' 'this' 'example'
```

-o option

If the code has been first tried out on command line, add `-o` option to get a pretty printed version. File name can be passed along `-o` option, otherwise `awkprof.out` will be used by default.

```
$ # adding -o after the one-liner has been tested
$ awk -o -v OFS='\t' 'NR==FNR{r[$1]=$2; next}
         {$(NF+1) = FNR==1 ? "Role" : r[$2]} 1' role.txt marks.txt

$ # pretty printed version
$ cat awkprof.out
NR == FNR {
        r[$1] = $2
        next
}

{
        $(NF + 1) = FNR == 1 ? "Role" : r[$2]
}

1 {
        print
}

$ # calling the script
$ # note that other command line options have to be provided as usual
$ awk -v OFS='\t' -f awkprof.out role.txt marks.txt
Dept    Name    Marks   Role
ECE     Raj     53      class_rep
ECE     Joel    72
EEE     Moi     68
CSE     Surya   81
EEE     Tia     59      placement_rep
ECE     Om      92
CSE     Amy     67      sports_rep
```

Summary

So, now you know how to write program files for `awk` instead of just the one-liners. And about the useful `-o` option, helps to convert complicated one-liners to pretty printed program files.

Next chapter will discuss a few gotchas and tricks.

Gotchas and Tips

Prefixing $ for variables

Some scripting languages like `bash` require a `$` prefix when you need the value stored in a variable. For example, if you declare `name='Joe'` you'd need `echo "$name"` to print the value. This may result in using `$` prefix and other bashisms in `awk` as well when you are a beginner. To make it a bit worse, `awk` has the `$N` syntax for accessing field contents, which could result in false comprehension that all variables need `$` prefix to access their values. See also unix.stackexchange: Why does awk print the whole line when I want it to print a variable?.

```
$ # silently fails, $word becomes $0 because of string to numeric conversion
$ awk -v word="cake" '$2==$word' table.txt
$ awk -v word="cake" '$2==word' table.txt
blue cake mug shirt -7

$ # here 'field' gets replaced with '2' and hence $2 is printed
$ awk -v field=2 '{print $field}' table.txt
bread
cake
banana
```

Dos style line endings

As mentioned before, line endings differ from one platform to another. On Windows, it is typically a combination of carriage return and the newline character and referred as **dos style** line endings. Since `GNU awk` allows multicharacter `RS` , it is easy to handle. See stackoverflow: Why does my tool output overwrite itself and how do I fix it? for a detailed discussion and various mitigation methods.

```
$ # no issue with unix style line ending
$ printf 'mat dog\n123 789\n' | awk '{print $2, $1}'
dog mat
789 123

$ # dos style line ending causes trouble
$ printf 'mat dog\r\n123 789\r\n' | awk '{print $2, $1}'
 mat
 123
$ printf 'mat dog\r\n123 789\r\n' | awk '{sub(/$/, ".")} 1'
.at dog
.23 789

$ # use \r?\n if you want to handle both unix and dos style with same command
$ # note that ORS would still be newline character only
$ printf 'mat dog\r\n123 789\r\n' | awk -v RS='\r\n' '{print $2, $1}'
dog mat
```

```
789 123
$ printf 'mat dog\r\n123 789\r\n' | awk -v RS='\r\n' '{sub(/$/, ".")} 1'
mat dog.
123 789.
```

Word boundary differences

The word boundary `\y` matches both start and end of word locations. Whereas, `\<` and `\>` match exactly the start and end of word locations respectively. This leads to cases where you have to choose which of these word boundaries to use depending on results desired. Consider `I have 12, he has 2!` as sample text, shown below as an image with vertical bars marking the word boundaries. The last character `!` doesn't have end of word boundary as it is not a word character.


```
$ # \y matches both start and end of word boundaries
$ # the first match here used starting boundary of 'I' and 'have'
$ echo 'I have 12, he has 2!' | awk '{gsub(/\y..\y/, "[&]")} 1'
[I ]have [12][, ][he] has[ 2]!

$ # \< and \> only match the start and end word boundaries respectively
$ echo 'I have 12, he has 2!' | awk '{gsub(/\<..\>/, "[&]")} 1'
I have [12], [he] has 2!
```

Here's another example to show the difference between the two types of word boundaries.

```
$ # add something to both start/end of word
$ echo 'hi log_42 12b' | awk '{gsub(/\y/, ":")} 1'
:hi: :log_42: :12b:

$ # add something only at start of word
$ echo 'hi log_42 12b' | awk '{gsub(/\</, ":")} 1'
:hi :log_42 :12b

$ # add something only at end of word
$ echo 'hi log_42 12b' | awk '{gsub(/\>/, ":")} 1'
hi: log_42: 12b:
```

Relying on default initial value

Uninitialized variables are useful, but sometimes they don't translate well if you are converting a command from single file input to multiple files. You have to workout which ones would need a reset at the beginning of each file being processed.

```
$ # step 1 - works for single file
$ awk '{sum += $NF} END{print sum}' table.txt
38.14
```

```
$ # step 2 - prepare code to work for multiple file
$ awk '{sum += $NF} ENDFILE{print FILENAME ":" sum}' table.txt
table.txt:38.14

$ # step 3 - check with multiple file input
$ # oops, default numerical value '0' for sum works only once
$ awk '{sum += $NF} ENDFILE{print FILENAME ":" sum}' table.txt marks.txt
table.txt:38.14
marks.txt:530.14

$ # step 4 - correctly initialize variables
$ awk '{sum += $NF} ENDFILE{print FILENAME ":" sum; sum=0}' table.txt marks.txt
table.txt:38.14
marks.txt:492
```

Forcing numeric context

Use unary operator `+` to force numeric conversion. A variable might have numeric operations but still not get assigned a number if there's no input to read. So, when printing a variable that should be a number, use unary `+` to ensure it prints `0` instead of empty string.

```
$ # numbers present in last column, no issues
$ awk '{sum += $NF} END{print sum}' table.txt
38.14
$ # strings in first column, gets treated as 0
$ awk '{sum += $1} END{print sum}' table.txt
0

$ # no input at all, empty string is printed
$ awk '{sum += $1} END{print sum}' /dev/null

$ # forced conversion to number, so that 0 is printed
$ awk '{sum += $1} END{print +sum}' /dev/null
0
```

Forcing string context

Concatenate empty string to force string comparison.

```
$ # parentheses around first argument to print used for clarity
$ # fields get compared as numbers here
$ echo '5 5.0' | awk '{print ($1==$2 ? "same" : "different"), "number"}'
same number

$ # fields get compared as strings here
$ echo '5 5.0' | awk '{print ($1""==$2 ? "same" : "different"), "string"}'
different string
```

Negative NF

Manipulating `NF` sometimes leads to a negative value. Fortunately, `awk` throws an error instead of behaving like uninitialized variable.

```
$ # file with different number of fields
$ cat varying.txt
parrot
good cool awesome
blue sky
12 34 56 78 90

$ # delete last two fields
$ awk '{NF -= 2} 1' varying.txt
awk: cmd. line:1: (FILENAME=varying.txt FNR=1) fatal: NF set to negative value

$ # add a condition to check number of fields
$ # assuming that lines with less than 3 fields should be preserved
$ awk 'NF>2{NF -= 2} 1' varying.txt
parrot
good
blue sky
12 34 56
```

Here's another example, which needs to access third field from the end.

```
$ awk '{print $(NF-2)}' varying.txt
awk: cmd. line:1: (FILENAME=varying.txt FNR=1) fatal: attempt to access field -1

$ # print only if there are minimum 3 fields
$ awk 'NF>2{print $(NF-2)}' varying.txt
good
56
```

Faster execution

Changing locale to ASCII (assuming current locale is not ASCII and the input file has only ASCII characters) can give significant speed boost.

```
$ # time shown is best result from multiple runs
$ # speed benefit will vary depending on computing resources, input, etc
$ # /usr/share/dict/words contains dictionary words, one word per line
$ time awk '/^([a-d][r-z]){3}$/' /usr/share/dict/words > f1
real    0m0.051s

$ time LC_ALL=C awk '/^([a-d][r-z]){3}$/' /usr/share/dict/words > f2
real    0m0.024s

$ # check that results are same for both versions of the command
$ diff -s f1 f2
```

```
Files f1 and f2 are identical
$ # clean up temporary files
$ rm f[12]
```

Here's another example.

```
$ # count words containing exactly 3 lowercase 'a'
$ time awk -F'a' 'NF==4{cnt++} END{print +cnt}' /usr/share/dict/words
1019
real    0m0.052s

$ time LC_ALL=C awk -F'a' 'NF==4{cnt++} END{print +cnt}' /usr/share/dict/words
1019
real    0m0.031s
```

Further Reading

- `man awk` and `info awk` and online manual
- Information about various implementations of `awk`
  - awk FAQ — great resource, but last modified *23 May 2002*
  - grymoire: awk tutorial — covers information about different `awk` versions as well
  - cheat sheet for awk/nawk/gawk
- Q&A on stackoverflow/stackexchange are good source of learning material, good for practice exercises as well
  - awk Q&A on unix.stackexchange
  - awk Q&A on stackoverflow
- Learn Regular Expressions (has information on flavors other than POSIX too)
  - regular-expressions — tutorials and tools
  - rexegg — tutorials, tricks and more
  - stackoverflow: What does this regex mean?
  - online regex tester and debugger — not fully suitable for cli tools, but most of the POSIX syntax works
- My repo on cli text processing tools
- Related tools
  - GNU datamash
  - bioawk
  - hawk — based on Haskell
  - miller — similar to awk/sed/cut/join/sort for name-indexed data such as CSV, TSV, and tabular JSON
    * See this news.ycombinator discussion for other tools like this
- miscellaneous
  - unix.stackexchange: When to use grep, sed, awk, perl, etc
  - awk-libs — lots of useful functions
  - awkaster — Pseudo-3D shooter written completely in awk
  - awk REPL — live editor on browser
- ASCII reference and locale usage
  - ASCII code table
  - wiki.archlinux: locale
  - shellhacks: Define Locale and Language Settings
- examples for some of the topics not covered in this book
  - unix.stackexchange: rand/srand
  - unix.stackexchange: strftime
  - stackoverflow: arbitrary precision integer extension
  - stackoverflow: recognizing hexadecimal numbers
  - unix.stackexchange: sprintf and file close
  - unix.stackexchange: user defined functions and array passing
  - unix.stackexchange: rename csv files based on number of fields in header row