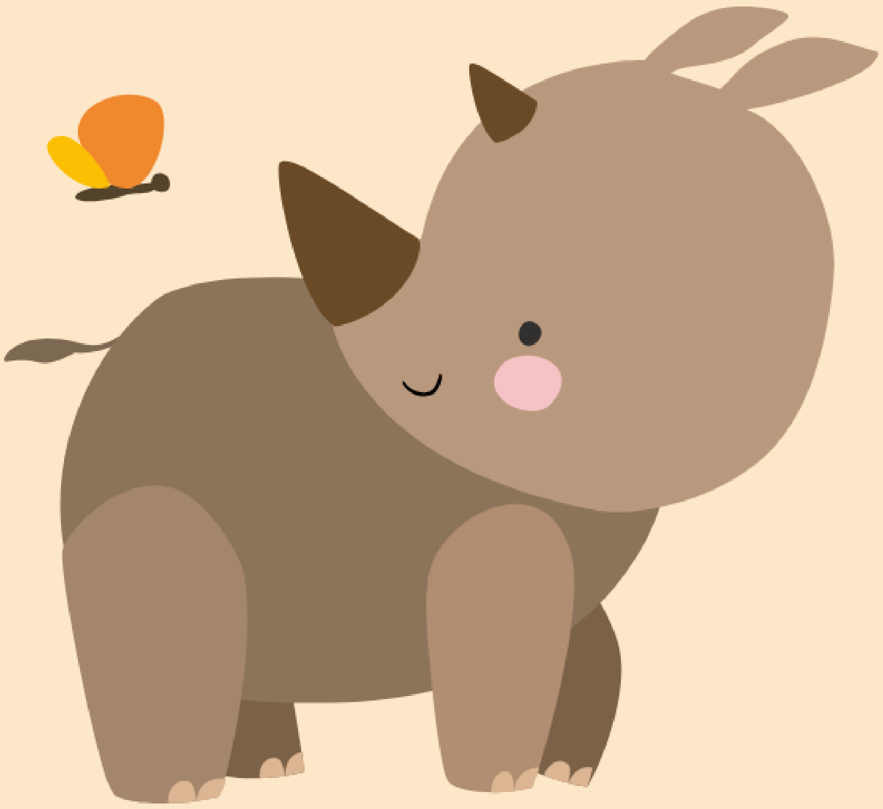
learnbyexample

CLI text processing with GNU awk



✓ 200+ examples

✓ 50+ exercises

Sundeep Agarwal

Table of contents

Preface

When it comes to command line text processing, the three major pillars are `grep` for filtering, `sed` for substitution and `awk` for field processing. These tools have overlapping features too, for example, all three of them have extensive filtering capabilities.

Unlike `grep` and `sed` , `awk` is a programming language. However, this book intends to showcase `awk` one-liners that can be composed from the command line instead of focusing on larger scripts.

This book heavily leans on examples to present features one by one. Regular expressions will also be discussed in detail.

It is recommended that you manually type each example. Make an effort to understand the sample input as well as the solution presented and check if the output changes (or not!) when you alter some part of the input and the command. As an analogy, consider learning to drive a car — no matter how much you read about them or listen to explanations, you'd need practical experience to become proficient.

Prerequisites

You should be familiar with command line usage in a Unix-like environment. You should also be comfortable with concepts like file redirection and command pipelines. Knowing the basics of the `grep` and `sed` commands will be handy in understanding the filtering and substitution features of `awk` .

As `awk` is a programming language, you are also expected to be familiar with concepts like variables, printing, functions, control structures, arrays and so on.

If you are new to the world of the command line, check out my Computing from the Command Line ebook and curated resources on Linux CLI and Shell scripting before starting this book.

Conventions

- The examples presented here have been tested with **GNU awk** version **5.2.2** and includes features not available in earlier versions.
- Code snippets are copy pasted from the `GNU bash` shell and modified for presentation purposes. Some commands are preceded by comments to provide context and explanations. Blank lines to improve readability, only `real` time shown for speed comparisons, output skipped for commands like `wget` and so on.
- Unless otherwise noted, all examples and explanations are meant for **ASCII** input.
- `awk` would mean `GNU awk` , `sed` would mean `GNU sed` , `grep` would mean `GNU grep` and so on unless otherwise specified.
- External links are provided throughout the book for you to explore certain topics in more depth.
- The learn_gnuawk repo has all the code snippets and files used in examples, exercises and other details related to the book. If you are not familiar with the `git` command, click the **Code** 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 `awk` and related commands
- tex.stackexchange — for help on pandoc and `tex` related questions
- /r/commandline/, /r/linux4noobs/, /r/linuxquestions/ and /r/linux/ — helpful forums
- canva — cover image
- oxipng, pngquant and svgcleaner — optimizing images
- Warning and Info icons by Amada44 under public domain
- arifmahmudrana for spotting an ambiguous explanation
- Pound-Hash for critical feedback

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 it if you'd let me know how you felt about this book. It could be anything from a simple thank you, pointing out a typo, mistakes in code snippets, which aspects of the book worked for you (or didn't!) and so on. Reader feedback is essential and especially so for self-published authors.

You can reach me via:

- Issue Manager: https://github.com/learnbyexample/learn_gnuawk/issues
- E-mail: learnbyexample.net@gmail.com
- Twitter: https://twitter.com/learn_byexample

Author info

Sundeep Agarwal is a lazy being who prefers to work just enough to support his modest lifestyle. He accumulated vast wealth working as a Design Engineer at Analog Devices and retired from the corporate world at the ripe age of twenty-eight. Unfortunately, he squandered his savings within a few years and had to scramble trying to earn a living. Against all odds, selling programming ebooks saved his lazy self from having to look for a job again. He can now afford all the fantasy ebooks he wants to read and spends unhealthy amount of time browsing the internet.

When the creative muse strikes, he can be found working on yet another programming ebook (which invariably ends up having at least one example with regular expressions). Researching materials for his ebooks and everyday social media usage drowned his bookmarks, so he maintains curated resource lists for sanity sake. He is thankful for free learning resources and open source tools. His own contributions can be found at https://github.com/learnbyexample.

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

License

Book version

2.0

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 will most likely have some version of `awk` already installed. This book is primarily about `GNU awk`. As there are syntax and feature differences between various implementations, make sure to use `GNU awk` to follow along the examples presented in this book.

`GNU awk` is part of the text creation and manipulation commands and usually comes by default on GNU/Linux distributions. To install a particular version, visit gnu: gawk software. See also release notes for an overview of changes between versions.

```
$ wget https://ftp.gnu.org/gnu/gawk/gawk-5.2.2.tar.xz
$ tar -Jxf gawk-5.2.2.tar.xz
$ cd gawk-5.2.2/
# see https://askubuntu.com/q/237576 if you get compiler not found error
$ ./configure
$ make
$ sudo make install

$ awk --version | head -n1
GNU Awk 5.2.2, API 3.2, PMA Avon 8-g1
```

If you are not using a Linux distribution, you may be able to access `GNU awk` using an option below:

- Git for Windows — provides a Bash emulation used to run Git from the command line
- Windows Subsystem for Linux — compatibility layer for running Linux binary executables natively on Windows
- brew — Package Manager for macOS (or Linux)

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

Documentation

It is always good to know where to find documentation. From the command line, you can use `man awk` for a short manual and `info awk` for the full documentation. I prefer using the online gnu awk manual, which feels much easier to use and navigate.

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 numerous GNU-specific extensions.
```

Options overview

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

```
$ 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
    -I                          --trace
    -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, but you shouldn't skip this chapter.

Filtering

`awk` provides filtering capabilities like those supported by the `grep` and `sed` commands. As a programming language, there are additional nifty features as well. 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'
# filter lines containing 'at'
$ printf 'gate\napple\nwhat\nkite\n' | awk '/at/'
gate
what

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

By default, `awk` automatically loops over the input content line by line. You can then use programming instructions to process those lines. As `awk` is often 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 the next chapter. String values without any special regexp characters are used in this chapter. 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. 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 the `;` character. You'll see such examples and learn more about `awk` syntax later.

Idiomatic use of 1

In a conditional expression, non-zero numeric values and non-empty string values are evaluated as `true`. 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 the `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 the 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 the `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 the 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 example_files directory has all the files used in the examples.

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

Here are some examples that are based on a specific field rather than the 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 are many more details and nuances to cover regarding the default field splitting, 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 the last field is a negative number
# recall that the default action is to print the contents of $0
$ awk '$NF<0' table.txt
blue cake mug shirt -7

# change 'b' to 'B' only for the 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 the previous sections have 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}'`

When a condition isn't provided, the action is always executed. Within a block, you can provide multiple statements separated by the semicolon character. If an 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 a 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 the input is read and an `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 an 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 for later use. Some variables are predefined, `NF` for example.

```
$ 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 an uninitialized variable is used, it will act as an 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 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 keys
$ 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

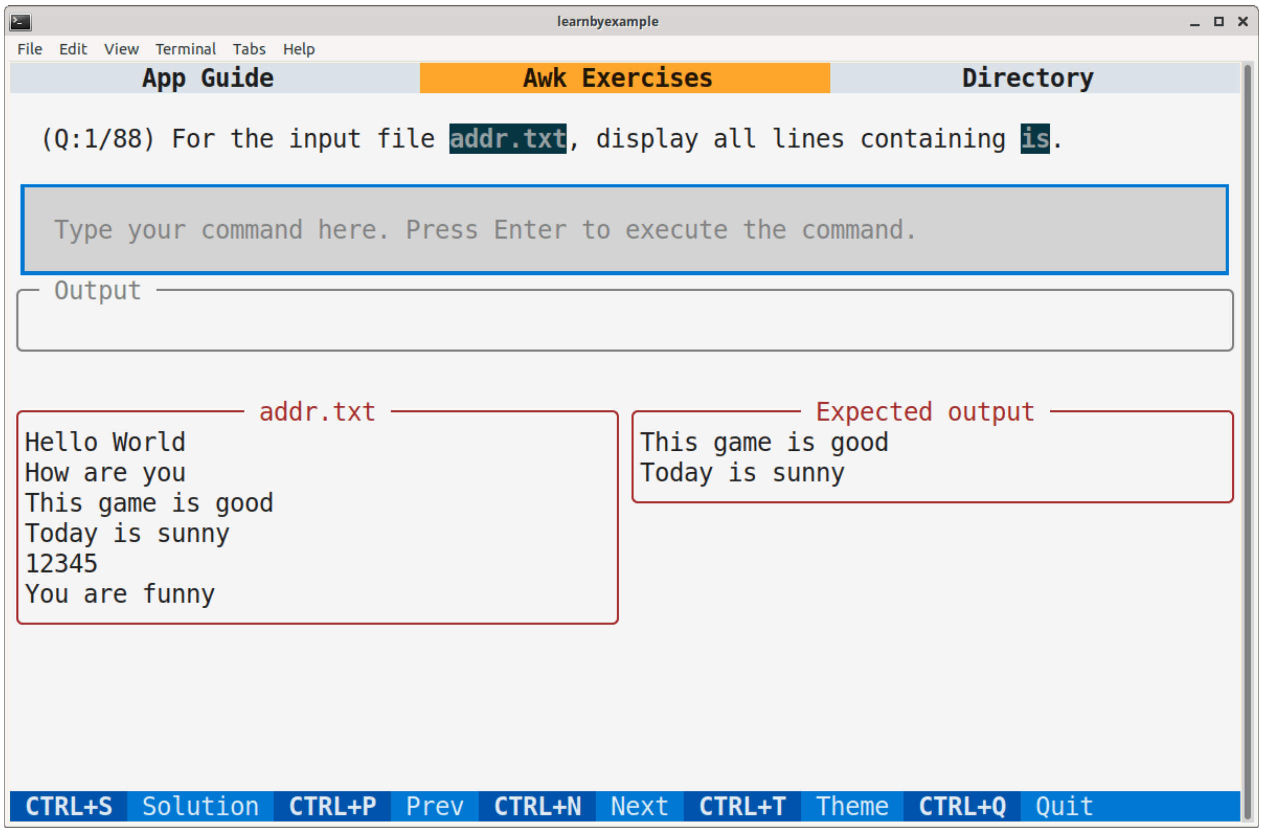
In my early days of getting used to the Linux command line, I was intimidated by `sed` and `awk` examples and didn't even try to learn them. Hopefully, this gentler introduction works for you and the various syntactical magic has been explained adequately. Try to experiment with the given examples, for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

The 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. Solving the exercises to follow will help test your understanding.

Interactive exercises

I wrote a TUI app to help you solve some of the exercises from this book interactively. See AwkExercises repo for installation steps and app_guide.md for instructions on using this app.

Here's a sample screenshot:


learnbyexample
File Edit View Terminal Tabs Help
App Guide
Awk Exercises
Directory
Type your command here. Press Enter to execute the command.
Output
CTRL+S Solution CTRL+P Prev CTRL+N Next CTRL+T Theme CTRL+Q Quit

(Q:1/88) For the input file addr.txt, display all lines containing is.

addr.txt

```
Hello World
How are you
This game is good
Today is sunny
12345
You are funny
```

Expected output

```
This game is good
Today is sunny
```

Exercises

All the exercises are also collated together in one place at Exercises.md. For solutions, see Exercise_solutions.md.

The exercises directory has all the files used in this section.

1) For the input file `addr.txt`, display all lines containing `is`.

```
$ cat addr.txt
Hello World
How are you
This game is good
Today is sunny
12345
You are funny

$ awk ##### add your solution here
This game is good
Today is sunny
```

2) For the input file `addr.txt`, display the first field of lines *not* containing `y`. Consider space as the field separator for this file.

```
$ awk ##### add your solution here
Hello
This
12345
```

3) For the input file `addr.txt`, display all lines containing no more than 2 fields.

```
$ awk ##### add your solution here
Hello World
12345
```

4) For the input file `addr.txt`, display all lines containing `is` in the second field.

```
$ awk ##### add your solution here
Today is sunny
```

5) For each line of the input file `addr.txt`, replace the first occurrence of `o` with `0`.

```
$ awk ##### add your solution here
Hell0 World
H0w are you
This game is g0od
T0day is sunny
12345
Y0u are funny
```

6) For the input file `table.txt`, calculate and display the product of numbers in the last field of each line. Consider space as the field separator for this file.

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

$ awk ##### add your solution here
-923.16
```

7) Append `.` to all the input lines for the given stdin data.

```
$ printf 'last\nappend\nstop\ntail\n' | awk ##### add your solution here
last.
append.
stop.
tail.
```

8) Replace all occurrences of `0xA0` with `0x50` and `0xFF` with `0x7F` for the given input file.

```
$ cat hex.txt
start address: 0xA0, func1 address: 0xA0
end address: 0xFF, func2 address: 0xB0

$ awk ##### add your solution here
start address: 0x50, func1 address: 0x50
end address: 0x7F, func2 address: 0xB0
```

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 are 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, digits, 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) supported by `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? See my blog post for differences between regexp features supported by `grep` , `sed` and `awk` .

The example_files directory has all the files used in the examples.

Syntax and variable assignment

As seen in the 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 adding `@` as a prefix. This 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 'BEGIN{r = @/ed/} $0 ~ r'
spared no one
grasped
```

String 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 string, 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 `\` character (discussed in the Matching the metacharacters section).

There are two string anchors:

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

By default, `awk` processes input line by line, using a newline character as the separator. This separator won't be part of the contents in `$0` but you get back the newline when printing because the default output record separator is also a newline character. Thus, these string anchors can be considered as *line* anchors when you are processing input content line by line.

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

# lines starting with 'sp'
$ awk '/^sp/' anchors.txt
spar

# lines ending with 'ar'
$ awk '/ar$/' anchors.txt
sub par
spar
```

By combining these two anchors, you can restrict the matching to only whole lines. Here's an example:

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

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

```
# add '* ' at the start of every input line
$ printf 'spared no one\ngrasped\nspar\n' | awk '{gsub(/^/, "* ")} 1'
* spared no one
* grasped
* spar
```

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

See also the Behavior of ^ and $ when string contains newline section.

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

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

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

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

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

Typically `\b` is used to represent the word anchor (for example, in `grep` , `sed` , `perl` , etc), but in `awk` the escape sequence `\b` refers to the backspace character. See also the Word boundary differences section.

Opposite Word Anchor

The `\y` escape sequence has an opposite anchor too. `\B` matches wherever `\y` doesn't match. This duality will be seen later with some other escape sequences too.

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

# match 'par' but not at the start of a word
$ awk '/\Bpar/' anchors.txt
spar
apparent effort
two spare computers

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

Here are some examples for using word boundaries by themselves as a pattern:

```
$ 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 the 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' and not containing 'at'
$ awk '/^b/ && !/at/' table.txt
blue cake mug shirt -7

# first field contains 'low'
# or, the last field value 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 (see the previous section for examples). With regular expressions, the `|` metacharacter is similar to logical OR. The regular expression will match if any of the patterns separated by `|` is satisfied.

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

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

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

Alternation precedence

There are some tricky corner cases 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.

```
# here, the output will be the same irrespective of alternation order
# note that 'sub' is used here, so only the 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 the same location, for example `spar` and `spared`, the longest matching portion gets precedence. Unlike other regular expression implementations, left-to-right priority for alternation comes into play only if the length of the matches are the same. See Longest match wins and Backreferences sections for more examples. See regular-expressions: alternation for more information on this topic.

```
$ echo 'spared party parent' | awk '{sub(/spa|spared/, "**")} 1'
** party parent
$ echo 'spared party parent' | awk '{sub(/spared|spa/, "**")} 1'
** party parent

# other regexp flavors like Perl have left-to-right priority
$ echo 'spared party parent' | perl -pe 's/spa|spared/**/'
**red 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
$ awk '/\<par\>|\<part\>/' anchors.txt
sub par
cart part tart mart
# taking out common anchors
$ awk '/\<(par|part)\>/' anchors.txt
sub par
cart part tart mart
# taking out common characters as well
# you'll later learn a better technique instead of using empty alternate
$ awk '/\<par(|t)\>/' anchors.txt
sub par
cart part tart mart
```

Matching the metacharacters

You have already seen a few metacharacters and escape sequences that help 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 string anchors have to be always escaped to match them literally as there is no BRE mode in `awk`. They do not lose their special meaning even when not used in their customary positions.

```
# awk '/b^2/' will not work even though ^ isn't being used as anchor
# b^2 will work for both grep and sed if you use BRE syntax
$ printf 'a^2 + b^2 - C*3\nd = c^2' | 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
```

Handling the replacement section metacharacters will be discussed in the Backreferences section.

Using string literal as a regexp

The first argument to the `sub` and `gsub` functions can be a string as well, which will then be converted to a regexp. This is handy in a few cases. For example, if you have many `/` characters in the search pattern, it might become easier to use a string literal instead of a regexp.

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

# filtering example
$ printf '/home/joe/1\n/home/john/1\n' | awk '/\/home\/joe\//'
/home/joe/1
$ printf '/home/joe/1\n/home/john/1\n' | awk '$0 ~ "/home/joe/"'
/home/joe/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 to represent regexp is that the special meaning for the `\` character will clash. For example:

```
$ awk 'gsub("\<par\>", "X")' 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 a single \
$ awk 'gsub("\\<par\\>", "X")' anchors.txt
sub X
# regexp literal is better suited in these cases
$ awk 'gsub(/\<par\>/, "X")' 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 to know.

The dot meta character

The dot metacharacter serves as a placeholder to match any character (including the newline character). Later you'll learn how to define your own custom placeholder for a 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 '42\t3500\n' | awk '{gsub(/.3./, ":")} 1'
42:00

# example to show that . matches \n as well
# 'c' followed by any character followed by 'x'
$ awk 'BEGIN{s="abc\nxyz"; sub(/c.x/, " ", s); print s}'
ab yz
```

Quantifiers

Alternation helps you match one among multiple patterns. Combining the dot metacharacter with quantifiers (and alternation if needed) paves a way to perform logical AND between patterns. For example, to check if a string matches two patterns with any number of characters in between. Quantifiers can be applied to characters, groupings and some more constructs that'll be discussed later. Apart from the 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.

```
# 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)\>/'
$ awk '/\<part?\>/' anchors.txt
sub par
cart part tart mart

# 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

# matches '<' or '\<' and they are both replaced with '\<'
$ echo 'apple \< fig ice < apple cream <' | awk '{gsub(/\\?</, "\\<")} 1'
apple \< fig ice \< apple cream \<
```

The `*` metacharacter quantifies a character or group to match `0` or more times.

```
# '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.

```
# '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

# 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:

Quantifier	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 stray characters like space are not allowed anywhere within {}
$ 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 the `()` metacharacters, escaping `{` alone is enough. If it doesn't conform strictly to any of the four forms listed above, escaping is not needed at all.

```
$ echo 'a{5} = 10' | awk '{sub(/a\{5}/, "x")} 1'
x = 10
$ echo 'report_{a,b}.txt' | awk '{sub(/_{a,b}/, "_c")} 1'
report_c.txt
```

Conditional AND

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.

```
# '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 where the longest matching portion was chosen if the alternatives started from the 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 the 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 '12apple' can match as well
$ echo 'fig123312apple' | awk '{sub(/g(1|2|3)+(12apple)?/, "X")} 1'
fiX
# in other implementations like Perl, that is not the case
# precedence is left-to-right for greedy quantifiers
$ echo 'fig123312apple' | perl -pe 's/g(1|2|3)+(12apple)?/X/'
fiXapple
```

While determining the longest match, the overall regular expression matching is also considered. That's how the `Error.*valid` example worked. If `.*` had consumed everything after `Error`, there wouldn't be any more characters to try to match `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 the first `a` in the input string to the last `b`. In other implementations, like Perl, this is achieved through a process called **backtracking**. These approaches have their own advantages and disadvantages and have cases where the pattern can result in exponential time consumption.

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

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

# from the first 'b' to the 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
# words.txt contains dictionary words, one word per line
$ awk '/^[on]{2,}$/' words.txt
no
non
noon
on
```

Character class metacharacters

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 a 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.

```
# 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 be 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. As highlighted earlier, handle negative logic with care, you might end up matching more than you wanted.

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

# delete last two columns
$ echo 'apple:123:banana:cherry' | awk '{sub(/(:[^:]+){2}$/, "")} 1'
apple:123

# sequence of characters surrounded by a 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

These escape sequences *cannot* be used inside character classes. Also, as mentioned earlier, these definitions assume ASCII input.

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

# \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.

```
# \d will match just the 'd' character and produces a warning as well
$ echo '42\d123' | awk '{gsub(/\d+/, "-")} 1'
awk: cmd. line:1: warning: regexp escape sequence
                  '\d' is not a known regexp operator
42\-123

# \d here matches all digit characters
$ echo '42\d123' | perl -pe 's/\d+/-/g'
-\d-
```

Named character sets

A named character set is defined by a name enclosed between `[:` and `:]` and has to be used within a character class `[]` , along with 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)
`[:punct:]`	all the punctuation characters
`[:graph:]`	`[:alnum:]` and `[:punct:]`

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

Here are some examples:

```
$ 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

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

Matching character class metacharacters literally

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 the first or the last character.

```
$ 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 the first character.

```
# no match
$ printf 'int a[5]\nfig\n1+1=2\n' | awk '/[=]]/'

# correct usage
$ printf 'int a[5]\nfig\n1+1=2\n' | awk '/[]=]/'
int a[5]
1+1=2
```

`[` can be used anywhere in the character set. Using `[][]` will match both `[` and `]` .

```
$ echo 'int a[5].y' | awk '{gsub(/[x[y.]/, "")} 1'
int a5]
```

```
$ printf 'int a[5]\nfig\n1+1=2\nwho]' | awk '/[][]/'
int a[5]
who]
```

`^` should be other than the first character.

```
$ 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.

```
# \t represents the tab character
$ printf 'apple\tbanana\tcherry\n' | awk '{gsub(/\t/, " ")} 1'
apple banana cherry

# these escape sequences 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
```

If a metacharacter is specified using the ASCII value format, it will still act as the metacharacter.

```
# \x5e is ^ character, acts as line anchor here
$ printf 'acorn\ncot\ncat\ncoat\n' | awk '/\x5eco/'
cot
coat

# & metacharacter in replacement will be discussed in a later section
# it represents the entire matched portion
$ echo 'hello world' | awk '{sub(/.*/, "[&]")} 1'
[hello world]
```

```
# \x26 in hexadecimal is the & character
$ echo 'hello world' | awk '{sub(/.*/, "[\x26]")} 1'
[hello world]
```

Undefined sequences will result in a warning and treated as the character it escapes.

```
$ echo 'read' | awk '{sub(/\d/, "l")} 1'
awk: cmd. line:1: warning: regexp escape sequence
                  '\d' is not a known regexp operator
real
```

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 the `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 a specific occurrence by passing a number. Another difference is that `gensub` returns a string value (irrespective of the substitution operation succeeding) instead of modifying the input.

```
$ s='apple:banana:cherry:fig:mango'

# same as: sed 's/:/-/2'
# replace only the second occurrence of ':' with '-'
# note that the output of gensub is passed to print here
$ echo "$s" | awk '{print gensub(/:/, "-", 2)}'
apple:banana-cherry:fig:mango

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

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

```
# 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**. Similar to variables in programming languages, the portion captured by `()` can be referred later using backreferences. The syntax is `\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`. The `&` metacharacter represents entire matched string. As `\` is already special inside double quotes, you'll have to use `"\\1"` to represent `\1`.

Backreferences of the form `\N` can only be used with the `gensub` function. `&` can be used with `sub`, `gsub` and `gensub` functions. `\0` can also be used instead of `&` with the `gensub` function.

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

# add something at the start and end of string, 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 the last but Nth occurrence
$ s='car,art,pot,tap,urn,ray,ear'
$ echo "$s" | awk '{print gensub(/(.*),((.*,){2})/, "\\1[]\\2", 1)}'
car,art,pot,tap[]urn,ray,ear
```

See unix.stackexchange: Why doesn't this sed command replace the 3rd-to-last "and"? for a bug related to the use of word anchors in the `((pat){N})` generic case.

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

```
$ s='effort flee facade oddball rat tool'

# no change
$ echo "$s" | awk '{gsub(/\w*(\w)\1\w*/, "X")} 1'
effort flee facade oddball rat tool
# whole words that have at least one consecutive repeated character
$ echo "$s" | sed -E 's/\w*(\w)\1\w*/X/g'
X X facade X rat X
```

If a quantifier is applied on a pattern grouped inside `()` metacharacters, you'll need an outer `()` group to capture the matching portion. Other flavors like Perl provide non-capturing groups to handle such cases. In `awk` you'll have to consider the extra capture groups.

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

Here's an example where alternation order matters when the matching portions have the same length. Aim is to delete all whole words unless it starts with `g` or `p` and contains `y`.

```
$ s='tryst,fun,glyph,pity,why,group'

# all words get deleted because \<\w+\> gets priority here
$ echo "$s" | awk '{print gensub(/\<\w+\>|(\<[gp]\w*y\w*\>)/, "\\1", "g")}'
,,,,,

# capture group gets priority here, so words in the capture group are retained
$ echo "$s" | awk '{print gensub(/(\<[gp]\w*y\w*\>)|\<\w+\>/, "\\1", "g")}'
,,glyph,pity,,
```

As `\` and `&` are special characters in the replacement section, you'll need to escape them for literal representation.

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

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

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 a 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 string, you can also use character class
$ printf 'Cat\ncOnCaT\nscatter\ncot\n' | awk '{gsub(/[cC][aA][tT]/, "(&)")} 1'
(Cat)
cOn(CaT)
s(cat)ter
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, string literals can be used instead of a 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 passed via the `-v` option 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 escape backslashes.

Sometimes, user input has to be treated literally instead of as a regexp pattern. In such cases, you'll need to escape all the regexp metacharacters. Below example shows how to do it for the search section. For the 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 the end of string
$ echo 'f*(a^b) - 3*(a^b)' |
    awk -v s='(a^b)' '{gsub(/[{[(^$*?+.|\\]/, "\\\\&", s); gsub(s "$", "c")} 1'
f*(a^b) - 3*c
```

See my blog post for more details about escaping metacharacters.

If you need to just match literally instead of substitution, you can use the `index` function. See the 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.

Exercises

The exercises directory has all the files used in this section.

1) For the input file `patterns.txt`, display all lines that start with `den` or end with `ly`.

```
$ awk ##### add your solution here
2 lonely
dent
lovely
```

2) For the input file `patterns.txt`, replace all occurrences of `42` with `[42]` unless it is at the edge of a word. Display only the modified lines.

```
$ awk ##### add your solution here
Hi[42]Bye nice1[42]3 bad42
eqn2 = pressure*3+42/5-1[42]56
cool_[42]a 42fake
_[42]_
```

3) For the input file `patterns.txt`, add `[]` around words starting with `s` and containing `e` and `t` in any order. Display only the modified lines.

```
$ awk ##### add your solution here
[sets] tests Sauerkraut
[site] cite kite bite [store_2]
[subtle] sequoia
a [set]
```

4) For the input file `patterns.txt`, replace the space character that occurs after a word ending with `a` or `r` with a newline character, only if the line also contains an uppercase letter. Display only the modified lines. For example, `A car park` should get converted to `A car` and `park` separated by a newline. But `car far tar` shouldn't be matched as there's no uppercase letter in this line.

```
$ awk ##### add your solution here
par
car
tar
far
Cart
Not a
pip DOWN
```

5) For the input file `patterns.txt`, replace all occurrences of `*[5]` with `2`. Display only the modified lines.

```
$ awk ##### add your solution here
(9-2)2
```

6) `awk '/\<[a-z](on|no)[a-z]\>/'` is same as `awk '/\<[a-z][on]{2}[a-z]\>/'` . True or False? Sample input shown below might help to understand the differences, if any.

```
$ printf 'known\nmood\nknow\npony\ninns\n'
known
mood
know
pony
inns
```

7) For the input file `patterns.txt` , display all lines starting with `hand` and ending immediately with `s` or `y` or `le` or no further characters. For example, `handed` shouldn't be matched even though it starts with `hand` .

```
$ awk ##### add your solution here
handle
handy
hands
hand
```

8) For the input file `patterns.txt` , replace `42//5` or `42/5` with `8` . Display only the modified lines.

```
$ awk ##### add your solution here
eqn3 = r*42-5/3+42///5-83+a
eqn1 = a+8-c
eqn2 = pressure*3+8-14256
```

9) For the given quantifiers, what would be the equivalent form using the `{m,n}` representation?

- `?` is same as
- `*` is same as
- `+` is same as

10) True or False? `(a*|b*)` is same as `(a|b)*`

11) For the input file `patterns.txt` , construct two different regexps to get the outputs as shown below. Display only the modified lines.

```
# delete from '(' till the next ')'
$ awk ##### add your solution here
a/b + c%d
*[5]
def factorial
12- *4)
Hi there. Nice day

# delete from '(' till the next ')' but not if there is '(' in between
$ awk ##### add your solution here
```

```
a/b + c%d
*[5]
def factorial
12- (e+*4)
Hi there. Nice day(a
```

12) For the input file `anchors.txt`, convert markdown anchors to corresponding hyperlinks as shown below.

```
$ cat anchors.txt
# <a name="regular-expressions"></a>Regular Expressions
## <a name="subexpression-calls"></a>Subexpression calls
## <a name="the-dot-meta-character"></a>The dot meta character

$ awk ##### add your solution here
[Regular Expressions](#regular-expressions)
[Subexpression calls](#subexpression-calls)
[The dot meta character](#the-dot-meta-character)
```

13) Display lines from `sample.txt` that satisfy both of these conditions:

- `to` or `he` matched irrespective of case
- `World` or `No` matched case sensitively

```
$ awk ##### add your solution here
Hello World
No doubt you like it too
```

14) Given sample strings have fields separated by `,` and field values cannot be empty. Replace the third field with `42`.

```
$ echo 'lion,ant,road,neon' | awk ##### add your solution here
lion,ant,42,neon

$ echo '_;3%,.,=-=,:' | awk ##### add your solution here
_;3%,.,42,:
```

15) For the input file `patterns.txt`, filter lines containing three or more occurrences of `ar` and replace the last but second `ar` with `X`.

```
$ awk ##### add your solution here
par car tX far Cart
pXt cart mart
```

16) Surround all whole words with `()`. Additionally, if the whole word is `imp` or `ant`, delete them.

```
$ words='tiger imp goat eagle ant important'
$ echo "$words" | awk ##### add your solution here
(tiger) () (goat) (eagle) () (important)
```

17) For the input file `patterns.txt`, display lines containing `car` but not as a whole word. For example, `scared-cat` and `car care` should match but not `far car park`.

```
$ awk ##### add your solution here
scar
care
a huge discarded pile of books
scare
part cart mart
```

18) Will the pattern `^a\w+([0-9]+:fig)?` match the same characters for the input `apple42:banana314` and `apple42:fig100` ? If not, why not?

19) For the input file `patterns.txt` , display lines starting with `4` or `-` or `u` or `sub` or `care` .

```
$ awk ##### add your solution here
care
4*5]
-handy
subtle sequoia
unhand
```

20) Replace sequences made up of words separated by `:` or `.` by the first word of the sequence. Such sequences will end when `:` or `.` is not followed by a word character.

```
$ ip='wow:Good:2_two.five: hi-2 bye kite.777:water.'
$ echo "$ip" | awk ##### add your solution here
wow hi-2 bye kite
```

21) Replace sequences made up of words separated by `:` or `.` by the last word of the sequence. Such sequences will end when `:` or `.` is not followed by a word character.

```
$ ip='wow:Good:2_two.five: hi-2 bye kite.777:water.'
$ echo "$ip" | awk ##### add your solution here
five hi-2 bye water
```

22) Replace all whole words with `X` unless it is preceded by a `(` character.

```
$ s='guava (apple) berry) apple (mango) (grape'
$ echo "$s" | awk ##### add your solution here
X (apple) X) X (mango) (grape
```

23) Surround whole words with `[]` only if they are followed by `:` or `,` or `-` .

```
$ ip='Poke,on=-=so_good:ink.to/is(vast)ever2-sit'
$ echo "$ip" | awk ##### add your solution here
[Poke],on=-=[so_good]:ink.to/is(vast)[ever2]-sit
```

24) The `fields.txt` file has fields separated by the `:` character. Delete `:` and the last field if there is a digit character anywhere before the last field.

```
$ cat fields.txt
42:cat
twelve:a2b
we:be:he:0:a:b:bother
apple:banana-42:cherry:
```

```
dragon:unicorn:centaur

$ awk ##### add your solution here
42
twelve:a2b
we:be:he:0:a:b
apple:banana-42:cherry
dragon:unicorn:centaur
```

25) Can you use a character other than `/` as the regexp delimiter? If not, are there ways to construct a regexp that do not require the `/` character to be escaped for literal matching?

26) For the input file `patterns.txt` , surround all hexadecimal sequences with a minimum of four characters with `[]` . Match `0x` as an optional prefix, but shouldn't be counted for determining the length. Match the characters case insensitively, and the sequences shouldn't be surrounded by other word characters. Display only the modified lines.

```
$ awk ##### add your solution here
"should not match [0XdeadBEEF]"
Hi42Bye nice1423 [bad42]
took 0xbad 22 [0x0ff1ce]
eqn2 = pressure*3+42/5-[14256]
```

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.

The example_files directory has all the files used in the examples.

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

# print the field as specified by the value stored in the '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 process fields 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!
# this will subtract 1 from the last field and print it
$ 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 the start or end of input gets trimmed and won't be part of the field contents. Input containing newline characters will be covered in the Record separators chapter.

```
$ echo '   a   b   c   ' | awk '{print NF}'
3
# note that the leading spaces aren't part of the field content
$ echo '   a   b   c   ' | awk '{print $1}'
a
# note that the trailing spaces aren't part of the 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 string literal and then converted to a regexp. For now, here are some examples without any special regexp characters.

```
# use ':' as the 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 "]"}'
[]

# difference between empty lines and lines without field separator
$ printf '\nhello\napple,banana\n' | awk -F, '{print NF}'
0
1
2
```

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 necessary here
$ 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 are some examples with regexp based field separators. The value passed to `-F` or `FS` is treated as a string and then converted to a 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 the number of vowels for each input line
# note that empty lines will give -1 in the output
$ printf 'cool\nnice car\n' | awk -F'[aeiou]' '{print NF-1}'
2
3
```

The default value of `FS` is a single space character. So, if you set the input field separator to a single space, then it will be the same as if you are using the default split discussed in the 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
```

If `IGNORECASE` is set, it will affect field separation as well. Except when the field separator is a single character, which can be worked around by using a character class.

```
$ echo 'RECONSTRUCTED' | awk -F'[aeiou]+' -v IGNORECASE=1 '{print $NF}'
D

# when FS is a single character
$ echo 'RECONSTRUCTED' | awk -F'e' -v IGNORECASE=1 '{print $1}'
RECONSTRUCTED
$ echo 'RECONSTRUCTED' | awk -F'[e]' -v IGNORECASE=1 '{print $1}'
R
```

Output field separator

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

```
# print the first and third fields, 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 are 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 the start/end gets trimmed for default FS
$ echo '   a   b   c   ' | awk '{$NF = "last"} 1'
a b last
```

Sometimes you want to print the 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 the 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
```

If you need to set the same input and output field separator, you can write a more concise one-liner using brace expansion. Here are some examples:

```
$ echo -v{,O}FS=:
-vFS=: -vOFS=:

$ echo 'goal:amazing:whistle:kwality' | awk -v{,O}FS=: '{$2 = 42} 1'
goal:42:whistle:kwality

$ echo 'goal:amazing:whistle:kwality' | awk '{$2 = 42} 1' {,O}FS=:
goal:42:whistle:kwality
```

However, this is not commonly used and doesn't save too many characters to be preferred over explicit assignment.

Manipulating NF

Changing the value of `NF` will rebuild `$0` as well. Here are some examples:

```
# 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

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

```
$ s='Sample123string42with777numbers'
# one or more consecutive digits
$ echo "$s" | awk -v FPAT='[0-9]+' '{print $2}'
42

$ s='coat Bin food tar12 best Apple fig_42'
# whole words made up of lowercase alphabets and digits only
$ echo "$s" | awk -v FPAT='\\<[a-z0-9]+\\>' -v OFS=, '{$1=$1} 1'
coat,food,tar12,best

$ s='items: "apple" and "mango"'
# get the first double quoted item
$ echo "$s" | awk -v FPAT='"[^"]+"' '{print $1}'
"apple"
```

`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? and csvquote 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.

A proper CSV support is planned for a future version. You can also check out frawk, which is mostly similar to the `awk` command but also supports CSV parsing. goawk is another implementation with CSV support.

If `IGNORECASE` is set, it will affect field matching as well. Unlike `FS` , there is no different behavior for a single character pattern.

```
# count number of 'e' in the input string
$ echo 'Read Eat Sleep' | awk -v FPAT='e' '{print NF}'
3
$ echo 'Read Eat Sleep' | awk -v IGNORECASE=1 -v FPAT='e' '{print NF}'
4
$ echo 'Read Eat Sleep' | awk -v IGNORECASE=1 -v FPAT='[e]' '{print NF}'
4
```

FIELDWIDTHS

`FIELDWIDTHS` is another feature where you get to define field contents. As indicated by the name, you have to specify the number of characters for each field. This method is useful to process fixed width data.

```
$ 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 the second field
# then 1 character is ignored and 6 characters for the 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 width 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 such 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 are many more examples to be discussed related to fields in the coming 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.

Exercises

The exercises directory has all the files used in this section.

1) For the input file `brackets.txt`, extract only the contents between `()` or `)(` from each input line. Assume that `()` characters will be present only once every line.

```
$ cat brackets.txt
foo blah blah(ice) 123 xyz$
(almond-pista) choco
yo )yoyo( yo

$ awk ##### add your solution here
ice
almond-pista
yoyo
```

2) For the input file `scores.csv` , extract `Name` and `Physics` fields in the format shown below.

```
$ cat scores.csv
Name,Maths,Physics,Chemistry
Blue,67,46,99
Lin,78,83,80
Er,56,79,92
Cy,97,98,95
Ort,68,72,66
Ith,100,100,100

$ awk ##### add your solution here
Name:Physics
Blue:46
Lin:83
Er:79
Cy:98
Ort:72
Ith:100
```

3) For the input file `scores.csv` , display names of those who've scored above `70` in Maths.

```
$ awk ##### add your solution here
Lin
Cy
Ith
```

4) Display the number of word characters for the given inputs. Word definition here is same as used in regular expressions. Can you construct a solution with `gsub` and one without substitution functions?

```
$ echo 'hi there' | awk ##### add your solution here
7

$ echo 'u-no;co%."(do_12:as' | awk ##### add your solution here
12
```

5) For the input file `quoted.txt` , extract the first and third sequence of characters surrounded by double quotes and display them in the format shown below. Solution shouldn't use substitution functions.

```
$ cat quoted.txt
1 "grape" and "mango" and "guava"
("a 1""b""c-2""d")

$ awk ##### add your solution here
"grape","guava"
"a 1","c-2"
```

6) For the input file `varying_fields.txt` , construct a solution to get the output shown below. Solution shouldn't use substitution functions.

```
$ cat varying_fields.txt
hi,bye,there,was,here,to
1,2,3,4,5

$ awk ##### add your solution here
hi,bye,to
1,2,5
```

7) Transform the given input file `fw.txt` to get the output as shown below. If a field is empty (i.e. contains only space characters), replace it with `NA` .

```
$ cat fw.txt
1.3  rs   90  0.134563
3.8           6
5.2  ye       8.2387
4.2  kt   32  45.1

$ awk ##### add your solution here
1.3,rs,0.134563
3.8,NA,6
5.2,ye,8.2387
4.2,kt,45.1
```

8) Display only the third and fifth characters from each input line as shown below.

```
$ printf 'restore\ncat one\ncricket' | awk ##### add your solution here
so
to
ik
```

9) The `fields.txt` file has fields separated by the `:` character. Delete `:` and the last field if there is a digit character anywhere before the last field. Solution shouldn't use substitution functions.

```
$ cat fields.txt
42:cat
twelve:a2b
we:be:he:0:a:b:bother
apple:banana-42:cherry:
dragon:unicorn:centaur

$ awk ##### add your solution here
42
twelve:a2b
we:be:he:0:a:b
apple:banana-42:cherry
dragon:unicorn:centaur
```

10) Retain only the first three fields for the given sample string that uses `^` as the input field separator. Use `,` as the output field separator.

```
$ echo 'sit^eat^very^eerie^near' | awk ##### add your solution here
sit,eat,very
```

11) The sample string shown below uses `cat` as the field separator (irrespective of case). Use space as the output field separator and add `42` as the last field.

```
$ s='applecatfigCaT12345cAtbanana'
$ echo "$s" | awk ##### add your solution here
apple fig 12345 banana 42
```

12) For the input file `sample.txt`, filter lines containing 6 or more lowercase vowels.

```
$ awk ##### add your solution here
No doubt you like it too
Much ado about nothing
```

13) The input file `concat.txt` has contents of various files preceded by a line starting with `###`. Replace such sequence of characters with an incrementing integer value (starting with `1`) in the format shown below.

```
$ awk ##### add your solution here
1) addr.txt
How are you
This game is good
Today is sunny
2) broken.txt
top
1234567890
bottom
3) sample.txt
Just do-it
Believe it
4) mixed_fs.txt
pink blue white yellow
car,mat,ball,basket
```

Record separators

So far, you've seen examples where `awk` automatically splits input line by line based on the 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 the `print` function. This chapter will also discuss how you can use special variables that have information related to record (line) numbers.

The example_files directory has all the files used in the examples.

Input record separator

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

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

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 `\n` , here's an example to show the full effect of the default record splitting.

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

Similar to `FS` , the `RS` value is treated as a string literal and then converted to a regexp. 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

# use 'Error:' as the input record separator
# print all the records that contains 'something'
```

```
$ awk -v RS='Error:' '/something/' report.log
 second record starts
something went wrong
some more details
```

If `IGNORECASE` is set, it will affect record separation as well. Except when the record separator is a single character, which can be worked around by using a character class.

```
$ awk -v IGNORECASE=1 -v RS='error:' 'NR==1' report.log
blah blah

# when RS is a single character
$ awk -v IGNORECASE=1 -v RS='e' 'NR==1' report.log
blah blah Error: s
$ awk -v IGNORECASE=1 -v RS='[e]' 'NR==1' report.log
blah blah
```

⚠ The default line ending for text files varies between different platforms. For example, a text file downloaded from the 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 to customize the 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 'apple\0banana\0cherry\0' | awk -v RS='\0' -v ORS='.\n' '1'
apple.
banana.
cherry.

$ cat msg.txt
Hello there.
It will rain to-
day. Have a safe
and pleasant jou-
rney.
# here ORS is an 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 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 the input doesn't end with a newline character
# 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 string literal and then converted to a regexp. Here are some examples.

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

# similar to FS, the value passed to RS is treated as a string
# which is then converted to a 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 an 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 the last record is a 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) [
]

# difference between FS and RS when they match till the end of the input
$ 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 the 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 an empty string, one or more consecutive empty lines is used as the input record separator. Consider the below sample file:

```
$ cat para.txt
Hello World

Hi there
How are you

Just do-it
Believe it

banana
papaya
mango

Much ado about nothing
He he he
Adios amigo
```

Here's an example of processing input paragraph wise:

```
# print all paragraphs containing 'do'
# note that there'll be an empty line after the last record
$ awk -v RS= -v ORS='\n\n' '/do/' para.txt
Just do-it
Believe it
```

```
Much ado about nothing
He he he
Adios amigo

```

The empty line at the end is a common problem when dealing with custom record separators. You could either process the output further to remove it or add logic to handle the issue in `awk` itself. Here's one possible workaround for the previous example:

```
# here ORS is left as the default newline character
# uninitialized variable 's' will be empty for the first match
# afterwards, 's' will provide the empty line separation
$ awk -v RS= '/do/{print s $0; s="\n"}' para.txt
Just do-it
Believe it

Much ado about nothing
He he he
Adios amigo
```

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 a 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 numbering. You've seen `NR` earlier in the chapter, but here are 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 the first field content only for the 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 a 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 are 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    0m0.395s
```

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 when 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.

Exercises

The exercises directory has all the files used in this section.

1) The input file `jumbled.txt` consists of words separated by various delimiters. Display all words that contain `an` or `at` or `in` or `it` , one per line.

```
$ cat jumbled.txt
overcoats;furrowing-typeface%pewter##hobby
wavering:concession/woof\retailer
joint[]seer{intuition}titanic

$ awk ##### add your solution here
overcoats
furrowing
wavering
joint
intuition
titanic
```

2) Emulate `paste -sd,` with `awk` .

```
# this command joins all input lines with the ',' character
$ paste -sd, addr.txt
Hello World,How are you,This game is good,Today is sunny,12345,You are funny
# make sure there's no ',' at end of the line
# and that there's a newline character at the end of the line
$ awk ##### add your solution here
Hello World,How are you,This game is good,Today is sunny,12345,You are funny

# if there's only one line in input, again make sure there's no trailing ','
$ printf 'fig' | paste -sd,
fig
$ printf 'fig' | awk ##### add your solution here
fig
```

3) For the input file `scores.csv`, add another column named **GP** which is calculated out of 100 by giving 50% weightage to Maths and 25% each for Physics and Chemistry.

```
$ awk ##### add your solution here
Name,Maths,Physics,Chemistry,GP
Blue,67,46,99,69.75
Lin,78,83,80,79.75
Er,56,79,92,70.75
Cy,97,98,95,96.75
Ort,68,72,66,68.5
Ith,100,100,100,100
```

4) For the input file `sample.txt`, extract paragraphs containing `do` and exactly two lines.

```
$ cat sample.txt
Hello World

Good day
How are you

Just do-it
Believe it

Today is sunny
Not a bit funny
No doubt you like it too

Much ado about nothing
He he he

# note that there's no extra empty line at the end of the output
$ awk ##### add your solution here
Just do-it
Believe it

Much ado about nothing
He he he
```

5) For the input file `sample.txt`, change each paragraph to a single line by joining lines using `.` and a space character as the separator. Also, add a final `.` to each paragraph.

```
# note that there's no extra empty line at the end of the output
$ awk ##### add your solution here
Hello World.

Good day. How are you.

Just do-it. Believe it.

Today is sunny. Not a bit funny. No doubt you like it too.

Much ado about nothing. He he he.
```

6) The various input/output separators can be changed dynamically and comes into effect during the next input/output operation. For the input file `mixed_fs.txt`, retain only the first two fields from each input line. The field separators should be space for the first two lines and `,` for the rest of the lines.

```
$ cat mixed_fs.txt
rose lily jasmine tulip
pink blue white yellow
car,mat,ball,basket
green,brown,black,purple
apple,banana,cherry

$ awk ##### add your solution here
rose lily
pink blue
car,mat
green,brown
apple,banana
```

7) For the input file `table.txt`, print other than the second line.

```
$ awk ##### add your solution here
brown bread mat hair 42
yellow banana window shoes 3.14
```

8) For the `table.txt` file, print only the line number for lines containing `air` or `win`.

```
$ awk ##### add your solution here
1
3
```

9) For the input file `table.txt`, calculate the sum of numbers in the last column, excluding the second line.

```
$ awk ##### add your solution here
45.14
```

10) Print the second and fourth line for every block of five lines.

```
$ seq 15 | awk ##### add your solution here
2
4
7
9
12
14
```

11) For the input file `odd.txt`, surround all whole words with `{}` that start and end with the same word character. This is a contrived exercise to make you use the `RT` variable (`sed -E 's/\b(\w)(\w*\1)?\b/{&}/g' odd.txt` would be a simpler solution).

```
$ cat odd.txt
-oreo-not:a _a2_ roar<=>took%22
RoaR to wow-
```

```
$ awk ##### add your solution here
-{oreo}-not:{a} {_a2_} {roar}<=>took%{22}
{RoaR} to {wow}-
```

12) Print only the second field of the third line, if any, from these input files: `addr.txt` , `sample.txt` and `copyright.txt` . Consider space as the field separator.

```
$ awk ##### add your solution here
game
day
bla
```

13) The input file `ip.txt` has varying amount of empty lines between the records, change them to be always two empty lines. Also, remove the empty lines at the start and end of the file.

```
$ awk ##### add your solution here
hello


world


apple
banana
cherry


tea coffee
chocolate
```

14) The sample string shown below uses `cat` as the record separator (irrespective of case). Display only the even numbered records separated by a single empty line.

```
$ s='applecatfigCaT12345cAtbananaCATguava:caT:mangocat3'
$ echo "$s" | awk ##### add your solution here
fig

banana

:mango
```

15) Input has the ASCII NUL character as the record separator. Change it to dot and newline characters as shown below.

```
$ printf 'apple\npie\0banana\ncherry\0' | awk ##### add your solution here
apple
pie.
banana
cherry.
```