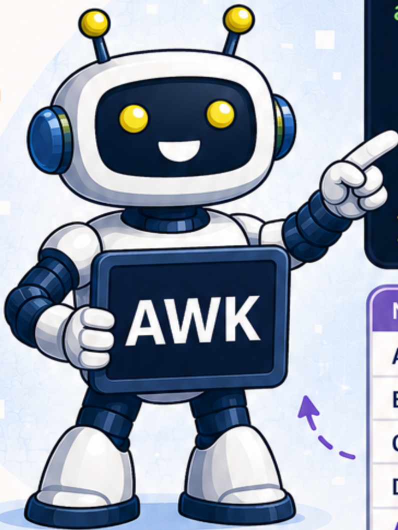
URADICAL

POCKET REFERENCE

```
awk 'NR>1 {
  sum += $3
  count++ }
  END {
   printf "Average: %.2f\n",
     sum/count
}' file.txt
```



Name	Dept	Salary
Alice	Eng	85000
Bob	Ops	72000
Carol	Eng	91000
Dave	Ops	68000
Average		79000

$0 FIELDS

NR / NF RECORDS

{ } PATTERNS

$1 $2 … ACTIONS

| PIPES

END END BLOCK

AWK

Alan Bradley

AWK Pocket Reference

A uRadical Production

Alan Bradley

uradical.io

AWK Pocket Reference

Alan Bradley

© 2026 uRadical

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

You are free to share and adapt this work for non-commercial purposes, as long as you give appropriate credit and distribute your contributions under the same license.

https://creativecommons.org/licenses/by-nc-sa/4.0/

Table of Contents

About This Book

Conventions

Chapter 1: The AWK Model

Chapter 2: Records and Fields

Chapter 3: Patterns

Chapter 4: Actions and Statements

Chapter 5: Variables and Expressions

Chapter 6: Regular Expressions

Chapter 7: Arrays

Chapter 8: Built-in Functions

Chapter 9: Output Redirection and Pipes

Chapter 10: AWK Programs in Files

Chapter 11: User-Defined Functions

Chapter 12: Error Handling and Defensive AWK

Chapter 13: Practical Recipes

Chapter 14: GNU AWK Extensions

Chapter 15: Debugging and Profiling

Chapter 16: Idioms and Patterns

Chapter 17: Locale and Encoding

Chapter 18: Gotchas and Edge Cases

Chapter 19: AWK vs Other Tools

Chapter 20: Cross-Platform Compatibility

Appendix A: One-Liner Cheat Sheet

Appendix B: Quick Reference Card

Appendix C: Task Index

Colophon

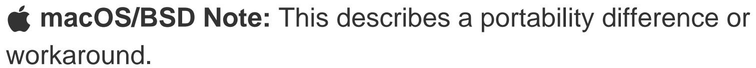
About This Book

AWK is one of the oldest and most powerful text-processing tools in the UNIX lineage. First written by Aho, Weinberger, and Kernighan at Bell Labs in 1977, it remains indispensable nearly five decades later. Where other tools have come and gone, AWK endures because it solves a permanent problem — structured text processing — with a minimal, composable design.

This pocket reference is written for GNU AWK (gawk), which ships as the default on Linux. Where a feature or behaviour does not work on macOS/BSD AWK, a compatibility note is provided so you can adapt. See Chapter 20 for a full portability reference.

Conventions

- `$` denotes a shell prompt.
- `awk` refers to GNU AWK unless a compatibility note says otherwise.
- Examples use single quotes for programs on the command line. If you are on a system where this causes problems, adjust accordingly.
- Filenames in examples use `data.txt`, `log.csv`, etc. Substitute your own.
- Compatibility callouts look like this:



> **macOS/BSD Note:** This describes a portability difference or workaround.

Chapter 1: The AWK Model

AWK is a pattern-action language. It reads input line by line, splits each line into fields, and executes actions when patterns match.

```
pattern { action }
```

If no pattern is given, the action runs on every line. If no action is given, the default is `{ print }`.

The Processing Cycle

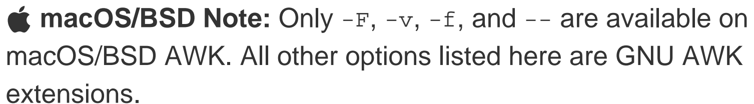
1. Read a record (by default, a line).
2. Split the record into fields (`$1`, `$2`, ..., `$NF`).
3. Evaluate each rule's pattern against the record.
4. Execute the action block for every matching pattern.
5. Repeat until end of input.

Invocation

```
# Inline program
$ awk 'program' file1 file2

# Program from file
$ awk -f script.awk file1

# With field separator
$ awk -F: 'program' /etc/passwd

# With variable assignment
$ awk -v threshold=100 '$3 > threshold' data.txt

# Read from stdin
$ cat data.txt | awk '{ print $2 }'
```

Command-Line Options

Option	Description
-F fs	Set field separator to fs
-v var=val	Assign variable before execution begins
-f file	Read program from file
--	End of options
-b	Binary mode
-i file	Include AWK source file
-l lib	Load shared library extension
--lint	Warn about dubious constructs
--posix	Disable all GNU extensions
--sandbox	Disable system(), I/O redirection, and |& — safe for untrusted input
--profile[=file]	Output execution profile



macOS/BSD Note: Only `-F`, `-v`, `-f`, and `--` are available on macOS/BSD AWK. All other options listed here are GNU AWK extensions.

Chapter 2: Records and Fields

Records

By default, records are lines separated by newlines. The built-in variable `NR` tracks the cumulative record number across all files. `FNR` tracks the record number within the current file.

```
# Print line numbers
$ awk '{ print NR, $0 }' data.txt
```

Change the record separator with `RS`:

```
# Paragraph mode — records separated by blank lines
$ awk 'BEGIN { RS="" } { print NR": "$0 }' essay.txt

# Single character
$ awk 'BEGIN { RS=";" } { print $0 }' data.csv

# Regex record separator
$ awk 'BEGIN { RS="\n---+\n" } { print NR, $0 }' doc.txt

# Slurp mode — read entire file as a single record
$ awk 'BEGIN { RS="\0" } { print length, "characters" }'
file.txt
```

Setting `RS` to an empty string enables paragraph mode (records separated by one or more blank lines). Setting `RS` to `"\0"` (the NUL character) effectively reads the entire file as one record, since NUL rarely appears in text files. This is useful when you need to match patterns that span multiple lines.

> **macOS/BSD Note:** Regex record separators do not work on macOS/BSD AWK. Only single-character `RS` values and the empty string (paragraph mode) are supported. If you set `RS` to a multi-character string, only the first character is used. The `"\0"` slurp trick works on both platforms since it is a single character.

Fields

Fields are numbered from `$1`. `$0` is the entire record. `$NF` is the last field. You can reference beyond the last field — it returns an empty string.

```
# Print first and third fields
$ awk '{ print $1, $3 }' data.txt

# Print the last field
$ awk '{ print $NF }' data.txt

# Print the second-to-last field
$ awk '{ print $(NF-1) }' data.txt
```

Field Separator

The field separator `FS` defaults to whitespace (spaces and tabs, with leading/trailing stripped). Set it with `-F` or in a `BEGIN` block.

```
# Colon-separated
$ awk -F: '{ print $1 }' /etc/passwd

# Multi-character separator
$ awk -F'::' '{ print $1 }' data.txt

# Regex separator
$ awk -F'[,;]' '{ print $1 }' data.txt

# Set in BEGIN
$ awk 'BEGIN { FS="," } { print $1 }' data.csv
```

Setting `FS` to a single space (the default) triggers special behaviour: leading and trailing whitespace is ignored, and runs of whitespace act as a single separator. Any other single character as `FS` does not have this behaviour — a leading separator will produce an empty `$1`.

Assigning Fields

You can assign to fields. This rebuilds `$0` using `OFS`.

```
# Replace the second field
$ awk '{ $2 = "REDACTED"; print }' data.txt

# Create new fields
$ awk '{ $(NF+1) = "new"; print }' data.txt
```

Output Separators

Variable	Default	Purpose
OFS	" " (space)	Output field separator (used when $0 is rebuilt)
ORS	"\n"	Output record separator

```
# Tab-separated output
$ awk 'BEGIN { OFS="\t" } { print $1, $3 }' data.txt

# Note: print $1, $3 uses OFS; print $1 $3 concatenates with
nothing
```

The distinction between `,` and concatenation in `print` is critical. `print $1, $3` inserts `OFS` between the two values. `print $1 $3` concatenates them directly with no separator.

Chapter 3: Patterns

Pattern Types

```
# Expression pattern — true when non-zero or non-empty
$3 > 100 { print }

# Regular expression — matches against $0
/error/ { print }

# Negated regex
!/^#/ { print }

# Field-specific regex match
$2 ~ /^[A-Z]/ { print }

# Negated field match
$4 !~ /test/ { print }

# Compound patterns
$3 > 50 && $4 == "active" { print }
$1 == "WARN" || $1 == "ERROR" { print }

# Range pattern — on from first match, off after second
/START/,/END/ { print }

# BEGIN and END — run before/after all input
BEGIN { print "Header" }
END { print "Done. Processed", NR, "lines." }
```

Range patterns are inclusive of both the start and end lines. If the end pattern is never matched, the range stays open until end of input. Overlapping ranges are not supported — once a range opens, AWK looks only for the end pattern, not for new start matches.

Truthiness

- The number `0` is false; all other numbers are true.
- The empty string `""` is false; all other strings are true.
- Uninitialised variables are `""` (and `0` in numeric context).

Chapter 4: Actions and Statements

Print and Printf

```
# print — outputs arguments separated by OFS, followed by ORS
{ print $1, $2, $3 }

# print with no arguments prints $0
/match/ { print }

# print to file
{ print $0 > "output.txt" }

# print appending to file
{ print $0 >> "log.txt" }

# print to command via pipe
{ print $0 | "sort -k2" }

# print to stderr
{ print "warning: bad input" > "/dev/stderr" }

# printf — formatted output (no trailing newline added)
{ printf "%-20s %6.2f\n", $1, $3 }
```

Printf Format Specifiers

Specifier	Description
%d, %i	Decimal integer
%f	Floating-point
%e, %E	Scientific notation
%g, %G	Shorter of %f and %e
%s	String
%c	Single character

%o	Octal
%x, %X	Hexadecimal
%%	Literal percent sign

Width and precision: `%10d` (right-aligned, width 10), `%-10s` (left-aligned), `%06.2f` (zero-padded, 2 decimal places).

Dynamic width and precision: Use `*` to take the width or precision from the next argument.

```
# Dynamic width — right-align to a variable column width
{ printf "%*s\n", width, $1 }

# Dynamic precision
{ printf "%.*f\n", decimals, $2 }

# Both
{ printf "%*.*f\n", width, decimals, $2 }
```

This is useful when you need to compute alignment at runtime — for instance, sizing columns to the longest value:

```
# First pass: find max width; second pass: print aligned
NR == FNR { if (length($1) > w) w = length($1); next }
{ printf "%-*s  %s\n", w, $1, $2 }
```

Control Flow

```
# if-else
{
    if ($3 > 100)
        print "high"
    else if ($3 > 50)
        print "medium"
    else
        print "low"
}
```

```
# while
{
    i = 1
    while (i <= NF) {
        print $i
        i++
    }
}

# do-while
{
    i = 1
    do {
        print $i
        i++
    } while (i <= NF)
}

# for
{
    for (i = 1; i <= NF; i++)
        print $i
}

# for-in (iterate over array keys — order is undefined)
END {
    for (key in counts)
        print key, counts[key]
}
```

Flow Control Keywords

Keyword	Effect
next	Stop processing current record, read next
nextfile	Skip to the next input file
exit	Jump to END block (or terminate if in END)
exit n	Set exit status to n

break	Exit innermost loop
continue	Skip to next iteration of innermost loop

 macOS/BSD Note: `nextfile` is POSIX 2008 and works on modern macOS AWK. If you are targeting very old systems, be aware it was originally a gawk extension.

Chapter 5: Variables and Expressions

Built-in Variables

Variable	Description
$0	Entire current record
$n	nth field of current record
NR	Total records read so far
NF	Number of fields in current record
FNR	Record number in current file
FS	Input field separator
RS	Input record separator
OFS	Output field separator
ORS	Output record separator
FILENAME	Current input filename
ARGC	Number of command-line arguments
ARGV	Array of command-line arguments
SUBSEP	Subscript separator for simulated multi-dimensional arrays (default \034)
RSTART	Start of string matched by match()
RLENGTH	Length of string matched by match()
ENVIRON	Array of environment variables
CONVFMT	Conversion format for implicit number-to-string conversion (default "%.6g")

OFMT	Output format for numbers in print (default "%.6g")
PROCINFO	Array of process and implementation info
ERRNO	Error string from last failed I/O operation
RT	The text matched by RS (useful when RS is a regex)

> **macOS/BSD Note:** `PROCINFO`, `ERRNO`, and `RT` are GNU AWK extensions and do not exist on macOS/BSD AWK. `ENVIRON` is supported on modern macOS AWK.

OFMT and CONVFMT — The Silent Precision Traps

`OFMT` controls how numbers are formatted when `print` outputs them. `CONVFMT` controls how numbers are converted to strings in other contexts (concatenation, array indexing, comparisons against strings). Both default to `"%.6g"`, which gives six significant digits.

This means AWK silently rounds large numbers and high-precision decimals:

```
# Large integers lose precision
BEGIN { id = 1234567890123; print id }
# Output: 1.23457e+12  — the last digits are gone

# Financial calculations lose decimals
BEGIN { price = 19.99; tax = 1.0825; print price * tax }
# Output: 21.6396  — OK here, but edge cases truncate

# Array keys lose precision via CONVFMT
{ arr[$1 + 0] = $2 }
# If $1 is "1234567890123", the key becomes "1.23457e+12"
```

If you work with large IDs, timestamps, or financial data, set these explicitly:

```
BEGIN {
    OFMT = "%.20g"
    CONVFMT = "%.20g"
}
```

Or use `printf` with an explicit format instead of `print`, which avoids `OFMT` entirely:

```
{ printf "%d\n", 1234567890123 }
# Output: 1234567890123
```

Operators

Arithmetic: `+`, `-`, `*`, `/`, `%` (modulo), `^` (exponent)

Assignment: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `^=`

Increment/Decrement: `++`, `--` (prefix and postfix)

Comparison: `==`, `!=`, `<`, `<=`, `>`, `>=`

Logical: `&&` (and), `||` (or), `!` (not)

String: concatenation is implicit (juxtaposition)

```
# Concatenation
{ fullname = $1 " " $2 }
```

Regex: `~` (matches), `!~` (does not match)

Ternary: `expr ? val_if_true : val_if_false`

```
{ status = ($3 > 0) ? "positive" : "non-positive" }
```

Unary plus: `+$1` forces numeric conversion — equivalent to `$1 + 0` but shorter.

Type Coercion

AWK is weakly typed. Variables hold both a string and a numeric value. Context determines which is used.

```
# Force numeric context
{ x = $1 + 0 }

# Force string context
{ s = $1 "" }
```

A common subtlety: comparing two variables that *look* like numbers produces a numeric comparison. Comparing a variable to a string constant always produces a string comparison. Comparing two variables that are both uninitialized compares the empty string to itself (equal).

```
# Numeric comparison — both look like numbers
{ if ($1 < $2) ... }

# String comparison — one side is a string constant
{ if ($1 < "10") ... }

# Force numeric regardless of context
{ if ($1 + 0 < $2 + 0) ... }
```

Chapter 6: Regular Expressions

AWK uses Extended Regular Expressions (ERE), the same flavour as `grep -E` and `egrep`. This is worth stating explicitly: if you are coming from `grep` or `sed` (which default to Basic Regular Expressions), the syntax differs.

ERE vs BRE — What Changes

Feature	BRE (grep, sed)	ERE (awk, grep -E)
Alternation	|	|
Grouping	\(\)	( )
One or more	\+	+
Zero or one	\?	?
Repetition	\{n,m\}	{n,m}
Literal parens	( )	\(\)

In AWK, you never need to escape `+`, `?`, `(`, `)`, `{`, or `}` for their special meanings. If you have been writing `sed` patterns, drop the backslashes.

Metacharacters

Character	Meaning
.	Any single character (except newline in most contexts)
^	Start of string (or start of record in pattern context)
$	End of string
*	Zero or more of preceding element

+	One or more of preceding element
?	Zero or one of preceding element
|	Alternation
( )	Grouping
[ ]	Character class (bracket expression)
{n}	Exactly n repetitions
{n,}	n or more repetitions
{n,m}	Between n and m repetitions
\	Escape the following character

Bracket Expressions

```
/[abc]/          # matches a, b, or c
/[a-z]/          # matches lowercase letters (locale-dependent
— see Chapter 17)
/[^0-9]/         # matches anything that is NOT a digit
/[a-zA-Z0-9_]/ # matches word characters (manually)
```

Literal special characters inside brackets: A literal `]` must be first: `[]abc]`. A literal `-` must be first or last: `[-abc]` or `[abc-]`. A literal `^` must not be first: `[a^b]`.

POSIX Character Classes

POSIX character classes are written inside an additional pair of brackets: `[[:class:]]`. They are locale-aware — their behaviour depends on the `LC_*` environment variables. See Chapter 17 for implications.

Class	Matches	Rough ASCII Equivalent
[[:alnum:]]	Alphanumeric	[a-zA-Z0-9]

[[:alpha:]]	Alphabetic	[a-zA-Z]
[[:blank:]]	Space and tab	[\t]
[[:cntrl:]]	Control characters	
[[:digit:]]	Digits	[0-9]
[[:graph:]]	Visible characters	[!-~]
[[:lower:]]	Lowercase	[a-z]
[[:print:]]	Printable (graph + space)	[-~]
[[:punct:]]	Punctuation	
[[:space:]]	Whitespace (space, tab, newline, CR, FF, VT)	[\t\n\r\f\v]
[[:upper:]]	Uppercase	[A-Z]
[[:xdigit:]]	Hex digits	[0-9a-fA-F]

The "rough ASCII equivalent" column is what you get in the `C`/`POSIX` locale. In a UTF-8 locale, `[[:alpha:]]` matches accented characters, Cyrillic, CJK, and so on — which is usually what you want but can cause surprises if you expect only ASCII.

Regex Contexts in AWK

Regexes appear in several places, and each behaves slightly differently:

```
# Pattern context — matches against $0
/regex/ { print }

# Match operator — matches a specific string
$2 ~ /regex/ { print }

# Negated match
$3 !~ /regex/ { print }

# In function arguments
match($0, /regex/)
```

```
sub(/regex/, "replacement")
gsub(/regex/, "replacement")
split($0, arr, /regex/)

# Dynamic regex from a variable
{ pattern = "^ERR"; if ($0 ~ pattern) print }
```

When a regex is stored in a variable as a string, AWK compiles it each time it is used. For performance-critical loops, this is measurable overhead.

Escape Sequences in Regex

Sequence	Meaning
\\	Literal backslash
\/	Literal forward slash (inside /.../)
\.	Literal dot
\n	Newline
\t	Tab
\r	Carriage return
\a	Alert (bell)
\b	Backspace (NOT a word boundary — AWK has no \b word boundary)

AWK does **not** support `\d`, `\w`, `\s`, `\b` (word boundary), or other Perl-style shorthand classes. Use POSIX classes instead: `[[:digit:]]` for `\d`, `[[:alnum:]_]` for `\w`, `[[:space:]]` for `\s`.

> **macOS/BSD Note:** The regex engine is POSIX ERE on both platforms and the syntax is the same. The differences are in locale handling of character classes (see Chapter 17) and in the GNU AWK extensions `gensub()` and `match(s, r, arr)` that allow backreferences and capture groups — not in the regex syntax

itself.

Chapter 7: Arrays

AWK arrays are associative (hash maps). All array indices are strings internally, regardless of how you write them.

```
# Basic usage
{ count[$1]++ }
END {
    for (key in count)
        print key, count[key]
}

# Testing membership (does NOT create the key)
if ("alice" in users)
    print "found"

# Deleting elements
delete arr["key"]

# Deleting entire array
delete arr
```

The distinction between `if (arr[x])` and `if (x in arr)` is important. The first creates the key `x` with an empty value if it does not exist, then tests its truthiness. The second tests membership without side effects. Always use `in` to test membership.

> **macOS/BSD Note:** `delete arr` (deleting an entire array without specifying a subscript) is implementation-defined in POSIX. It works in modern macOS AWK, but on truly ancient systems you may need to loop: `for (k in arr) delete arr[k]`.

Simulated Multi-Dimensional Arrays

AWK simulates multi-dimensional arrays by concatenating indices with `SUBSEP` (default `\034`, the ASCII file separator character).

```
{ grid[$1, $2] = $3 }

# Test membership
if ((x, y) in grid)
    print grid[x, y]

# Iterate
for (pair in grid) {
    split(pair, idx, SUBSEP)
    print idx[1], idx[2], grid[idx[1], idx[2]]
}
```

True Multi-Dimensional Arrays

GNU AWK supports true arrays of arrays, where each element can itself be an array.

```
{ data[$1][$2] = $3 }

END {
    for (row in data)
        for (col in data[row])
            print row, col, data[row][col]
}
```

> **macOS/BSD Note:** True multi-dimensional arrays (`arr[i][j]`) are a GNU AWK extension and will produce a syntax error on macOS/BSD AWK. Use the SUBSEP simulation (`arr[i, j]`) shown above for portable code.

Sorting Arrays

GNU AWK can control `for-in` traversal order via `PROCINFO["sorted_in"]`.

```
END {
    PROCINFO["sorted_in"] = "@val_num_desc"
    for (key in counts)
        print key, counts[key]
```

```
}
```

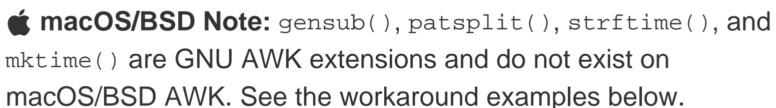
Sort orders: `@ind_str_asc`, `@ind_str_desc`, `@ind_num_asc`, `@ind_num_desc`, `@val_str_asc`, `@val_str_desc`, `@val_num_asc`, `@val_num_desc`, `@unsorted`.

> **macOS/BSD Note:** `PROCINFO` and sorted traversal are GNU AWK only. On macOS/BSD, pipe to `sort` instead:
>
> ```
> awk '{ count[$1]++ } END { for (k in count) print count[k], k }' file | sort -rn
> ```

Chapter 8: Built-in Functions

String Functions

Function	Description
length(s)	Length of string s (or $0 if omitted)
substr(s, start [, len])	Substring starting at start (1-indexed)
index(s, target)	Position of target in s (0 if not found)
split(s, arr [, fs])	Split s into arr using fs; returns count
sub(regex, replacement [, target])	Replace first match in target (default $0)
gsub(regex, replacement [, target])	Replace all matches; returns replacement count
match(s, regex)	Find regex in s; sets RSTART, RLENGTH; returns RSTART or 0
sprintf(fmt, ...)	Return formatted string (does not print)
tolower(s)	Lowercase
toupper(s)	Uppercase
gensub(regex, replacement, how [, target])	Like gsub but returns a new string and supports \1 backreferences. how is "g" or a count.
patsplit(s, arr, regex [, seps])	Split by regex matches (not separators)
strftime(fmt [, timestamp])	Format a timestamp
mktime(datespec)	Convert date string to timestamp



macOS/BSD Note: `gensub()`, `patsplit()`, `strftime()`, and `mktime()` are GNU AWK extensions and do not exist on macOS/BSD AWK. See the workaround examples below.

```
# sub and gsub — use & in replacement to reference matched
text
{ sub(/foo/, "bar") }           # first occurrence in $0
{ gsub(/[[:space:]]+/, " ") }  # collapse whitespace
{ sub(/.*/, "(&)") }            # wrap entire line in
parentheses

# match
{
    if (match($0, /[0-9]+/))
        print substr($0, RSTART, RLENGTH)
}

# split
{
    n = split($0, parts, ",")
    for (i = 1; i <= n; i++)
        print parts[i]
}
```

sprintf — The General-Purpose String Builder

`sprintf` is one of AWK's most versatile functions. It works exactly like `printf` but returns the formatted string instead of printing it. This makes it essential for constructing filenames, array keys, padded values, and any string that needs structure.

```
# Construct output filenames
{
    outfile = sprintf("output/%s_%04d.txt", $1, NR)
    print $0 > outfile
}

# Build zero-padded keys for correct sort order
{
    key = sprintf("%05d", $1)
    data[key] = $0
}

# Construct composite array keys with controlled formatting
{
```

```
    key = sprintf("%s|%s|%04d", $1, $2, $3)
    seen[key]++
}

# Format numbers for display without printing
{
    formatted_price = sprintf("£%0.2f", $3)
    formatted_pct = sprintf("%5.1f%%", $4)
    print $1, formatted_price, formatted_pct
}

# Build fixed-width report lines
{
    line = sprintf("%-30s %10.2f %8d", $1, $2, $3)
    if ($2 > threshold)
        line = line "  ***"
    print line
}

# Pad strings to a fixed width
function pad(s, w) {
    return sprintf("%-*s", w, s)
}
```

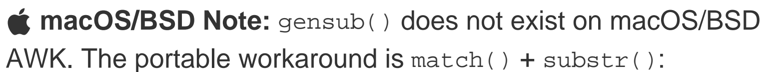
`sprintf` is often preferable to string concatenation because it gives you explicit control over numeric formatting, padding, and alignment — all without the `OFMT`/`CONVFMT` precision traps that affect implicit conversion (see Chapter 5).

Replacement string rules for `sub` and `gsub`: The character `&` in the replacement is replaced by the text that matched the regex. To get a literal `&`, use `\\&`. To get a literal backslash, use `\\\\`. This double-escaping is a frequent source of bugs.

gensub — and Its Portable Alternative

`gensub` is the most common source of cross-platform breakage. It is the only AWK string function that supports backreferences (`\1`, `\2`, etc.) and returns the modified string without altering the original.

```
# gensub — swap first two words
{ print gensub(/^([^ ]+) ([^ ]+)/, "\\2 \\1", 1) }

# gensub — extract domain from email
{ domain = gensub(/.*@(.*)/, "\\1", 1, $2) }

# gensub — replace nth occurrence
{ print gensub(/pattern/, "replacement", 3) }  # third match only
```



macOS/BSD Note: `gensub()` does not exist on macOS/BSD AWK. The portable workaround is `match()` + `substr()`:

```
# Portable equivalent — swap first two words
{
    if (match($0, /^[^ ]+/)) {
        first = substr($0, RSTART, RLENGTH)
        rest = substr($0, RSTART + RLENGTH)
        if (match(rest, /^ +/)) {
            gap = substr(rest, RSTART, RLENGTH)
            after_gap = substr(rest, RSTART + RLENGTH)
            if (match(after_gap, /^[^ ]+/)) {
                second = substr(after_gap, RSTART, RLENGTH)
                tail = substr(after_gap, RSTART + RLENGTH)
                print second gap first tail
            }
        }
    }
}
```

For complex regex replacements requiring backreferences, consider piping through `sed -E` or restructuring the logic to avoid them.

match() with Capture Groups

GNU AWK extends `match()` to accept a third argument — an array that receives captured groups.

```
{
    if (match($0, /([0-9]+)-([A-Z]+)/, arr))
        print "Number:", arr[1], "Code:", arr[2]
}

# Extract key=value pairs
{
    if (match($0, /([^=]+)=(.*)/, kv))
        config[kv[1]] = kv[2]
}
```

> **macOS/BSD Note:** The third argument to `match()` is a GNU AWK extension. On macOS/BSD, use repeated `match()` and `substr()` calls to extract sub-parts of a match.

Arithmetic Functions

Function	Description
int(x)	Truncate to integer (towards zero)
sqrt(x)	Square root
exp(x)	Exponential (e^x)
log(x)	Natural logarithm
sin(x)	Sine (radians)
cos(x)	Cosine (radians)
atan2(y, x)	Arctangent of y/x
rand()	Pseudo-random number in [0, 1)
srand([seed])	Seed the random number generator; returns previous seed

```
# Random integer between 1 and 6
BEGIN { srand() }
{ print int(rand() * 6) + 1 }
```

```
# Round to 2 decimal places
{ print int($1 * 100 + 0.5) / 100 }

# Ceiling function (AWK has no built-in ceil)
function ceil(x) { return (x == int(x)) ? x : int(x) + 1 }

# Floor function
function floor(x) { return int(x) }
```

Note that `int()` truncates towards zero: `int(3.9)` is `3`, `int(-3.9)` is `-3`. If you want consistent floor behaviour for negative numbers, use: `function floor(x) { return (x >= 0 || x == int(x)) ? int(x) : int(x) - 1 }`.

I/O Functions

Function	Description
getline	Read next record from current input into $0
getline var	Read next record into var (does not alter $0)
getline < file	Read from file into $0
getline var < file	Read from file into var
command | getline	Read from command output into $0
command | getline var	Read from command output into var
close(file_or_command)	Close a file or pipe
system(command)	Execute shell command; returns exit status
fflush([file])	Flush output buffer

Getline return values: `1` (success), `0` (end of input), `-1` (error). Always check the return value in a conditional or loop.

```
# Read a lookup file into an array — ALWAYS check return value
BEGIN {
    while ((getline line < "lookup.txt") > 0) {
        split(line, parts, ",")
        lookup[parts[1]] = parts[2]
    }
    close("lookup.txt")
}

# Get output from a command
{
    cmd = "date -d \"" $1 "\" +%s"
    if ((cmd | getline epoch) > 0)
        print $0, epoch
    else
        print $0, "DATE_ERROR"
    close(cmd)
}
```

The `getline` family is powerful but has surprising side effects. A bare `getline` (no variable) overwrites `$0` and recomputes all fields. `getline var` only sets `var` and `NR`/`FNR`. `getline < file` sets `$0` but does not update `NR` or `FNR`. Knowing which side effects each form has prevents subtle bugs.

Form	Sets $0?	Sets fields?	Updates NR?	Updates FNR?
getline	Yes	Yes	Yes	Yes
getline var	No	No	Yes	Yes
getline < file	Yes	Yes	No	No
getline var < file	No	No	No	No
cmd | getline	Yes	Yes	No	No
cmd | getline var	No	No	No	No

 macOS/BSD Note: The `date -d` syntax used above is GNU coreutils. On macOS, use `date -j -f` instead. This is a coreutils difference, not an AWK difference, but it frequently bites people in AWK scripts that shell out to `date`.

Chapter 9: Output Redirection and Pipes

Writing to Files

```
# Write to file — truncated on first write, then appended
within the run
{ print $0 > "output.txt" }

# Append to file
{ print $0 >> "append.txt" }
```

A common misconception: `>` does not truncate on every write. It truncates once (the first time AWK opens the file in a given run), then all subsequent `>` writes to the same filename append. This means you do not need `>>` to accumulate output during a single AWK invocation.

Writing to stderr

```
# Always use /dev/stderr for diagnostic output
{ print "WARNING: invalid line " NR > "/dev/stderr" }
```

This keeps diagnostic messages separate from `stdout`, which is essential when your AWK output is piped to another command.

Pipes

```
# Pipe to command
{ print $0 | "sort" }

# Pipe to command — same string = same pipe
{ print $1 | "sort -u" }
```

AWK keeps files and pipes open until you `close()` them or the program ends. The filename or command string is the handle — if you use the exact same string, it refers to the same open file or pipe. If you change a single character in the command string, AWK opens a new, separate

pipe.

```
# Close and reopen to get fresh output
{
    print $0 | "sort"
}
END {
    close("sort")
}
```

Two-Way Pipes (Coprocess)

```
{ print "query" |& "program"; "program" |& getline response }
```

> **macOS/BSD Note:** The two-way pipe operator `|&` is a GNU AWK extension and does not exist on macOS/BSD AWK. There is no direct workaround within AWK. Use temporary files or restructure your pipeline at the shell level:
>
> ```
> awk '{ print $1 }' data.txt > /tmp/awk_input.$$
> program < /tmp/awk_input.$$ > /tmp/awk_output.$$
> awk '{ ... }' /tmp/awk_output.$$
> rm -f /tmp/awk_input.$$ /tmp/awk_output.$$
> ```

Chapter 10: AWK Programs in Files

For anything beyond a one-liner, put your AWK program in a file.

A Simple Report

```
#!/usr/bin/awk -f
# report.awk — summarise sales data

BEGIN {
    FS = ","
    printf "%-20s %10s %8s\n", "Product", "Revenue", "Count"
    printf "%-20s %10s %8s\n", "-------", "-------", "-----"
}

NR > 1 {
    revenue[$1] += $3
    count[$1]++
    total += $3
}

END {
    for (product in revenue)
        printf "%-20s %10.2f %8d\n", product,
revenue[product], count[product]

    printf "\n%-20s %10.2f %8d\n", "TOTAL", total, NR-1
}
```

```
$ chmod +x report.awk
$ ./report.awk sales.csv
# or
$ awk -f report.awk sales.csv
```

> **macOS/BSD Note:** The shebang `#!/usr/bin/awk -f` works on both platforms. On macOS, this invokes the system AWK. If you have installed gawk via Homebrew, use `#!/usr/local/bin/gawk -f` or `#!/opt/homebrew/bin/gawk -f` (Apple Silicon) instead. For maximum portability across systems, `#!/usr/bin/env awk -f`

seems attractive but is unreliable — POSIX does not require `env` to split arguments after the program name, so some kernels pass `"awk -f"` as a single argument to `env`, which fails. Linux handles this correctly; older FreeBSD and some embedded systems do not.

A Longer Example: Log Analyser

This example demonstrates how functions, multiple rules, defensive validation, and a state machine compose in a real script. It processes an HTTP access log and produces a summary grouped by hour, flagging anomalies.

```
#!/usr/bin/awk -f
# logwatch.awk — HTTP access log analyser
#
# Usage: awk -f logwatch.awk -v error_threshold=5 access.log
# Default error threshold: 10%

BEGIN {
    if (!error_threshold) error_threshold = 10

    # Column widths for output
    hdr_fmt = "%-6s  %8s  %8s  %8s  %7s  %s\n"
    row_fmt = "%-6s  %8d  %8d  %8d  %6.1f%%  %s\n"

    printf hdr_fmt, "Hour", "Requests", "2xx", "4xx/5xx",
"Err%", "Status"
}

# Skip blank lines and malformed records
NF < 9 { next }

# Extract hour from CLF timestamp: [18/May/2026:14:23:01
+0000]
{
    if (!match($4, /:[0-9][0-9]:/)) {
        warn("cannot parse timestamp", $4)
        next
    }
    hour = substr($4, RSTART+1, 2)
```

```
}

# Validate status code is numeric
$9 !~ /^[0-9]+$/ {
    warn("non-numeric status code", $9)
    next
}

# Accumulate
{
    requests[hour]++
    status = int($9)

    if (status >= 200 && status < 300)
        ok[hour]++
    else if (status >= 400)
        errors[hour]++

    total_requests++
}

END {
    # Sorted output: hours 00-23
    for (h = 0; h < 24; h++) {
        hh = sprintf("%02d", h)
        if (!(hh in requests)) continue

        req = requests[hh]
        good = (hh in ok) ? ok[hh] : 0
        bad = (hh in errors) ? errors[hh] : 0
        err_pct = (req > 0) ? (bad / req * 100) : 0
        flag = (err_pct > error_threshold) ? ". HIGH ERRORS" :
"OK"

        printf row_fmt, hh ":00", req, good, bad, err_pct,
flag
    }

    printf "\n"
    printf "Total requests: %d\n", total_requests
    printf "Error threshold: %d%%\n", error_threshold
}
```

```
function warn(msg, detail) {
    printf "WARN [line %d]: %s: %s\n", NR, msg, detail >
"/dev/stderr"
}
```

This script demonstrates several patterns working together: a `BEGIN` block that handles defaults and prints headers, input validation with `next` to skip bad records, a utility function for consistent diagnostic output, manual sorted iteration (hours 00-23) without relying on `PROCINFO`, and a `-v` variable for runtime configuration.

Structuring Larger Programs

- Use `BEGIN` for setup: assign `FS`, initialise variables, print headers, validate `ARGC`.
- Use multiple pattern-action rules for clarity — each handles one concern.
- Use `END` for summaries and cleanup.
- Use functions to factor out repeated logic and to hold local variables.
- Keep rules in a logical order: `BEGIN`, then data-loading rules, then processing rules, then `END`.

Chapter 11: User-Defined Functions

```
function abs(x) {
    return (x < 0) ? -x : x
}

function max(a, b) {
    return (a > b) ? a : b
}

function min(a, b) {
    return (a < b) ? a : b
}

function trim(s) {
    gsub(/^[[:space:]]+|[[:space:]]+$/, "", s)
    return s
}

function ltrim(s) {
    sub(/^[[:space:]]+/, "", s)
    return s
}

function rtrim(s) {
    sub(/[[:space:]]+$/, "", s)
    return s
}

function join(arr, n, sep,    result, i) {
    # Parameters after the extra spaces are local variables
    result = arr[1]
    for (i = 2; i <= n; i++)
        result = result sep arr[i]
    return result
}

function repeat(s, n,    result, i) {
    result = ""
    for (i = 1; i <= n; i++)
        result = result s
```

```
    return result
}

{ print trim($1), abs($3) }
```

Local variables: AWK has no `local` keyword. By convention, extra parameters in the function signature (separated by extra whitespace from the real parameters) serve as local variables. This is ugly but universal — every AWK implementation supports it.

Recursion works normally. Each invocation gets its own copy of the parameters and local variables. Be aware that AWK has no tail-call optimisation, so deep recursion will exhaust the stack.

Functions must be defined at the top level, outside of any pattern-action rules. They can be placed before or after the rules — AWK parses the entire program before execution begins.

Chapter 12: Error Handling and Defensive AWK

AWK programs often process data of uncertain quality. Defensive patterns prevent silent failures.

Guarding getline

Every `getline` call should check its return value.

```
# Bad — silently ignores errors
getline line < "config.txt"

# Good — handles errors explicitly
if ((getline line < "config.txt") <= 0) {
    print "ERROR: cannot read config.txt" > "/dev/stderr"
    exit 1
}
close("config.txt")
```

Guarding Division

```
# Bad — division by zero produces an error or inf
{ print $1 / $2 }

# Good
{ print ($2 != 0) ? $1 / $2 : "N/A" }
```

Validating Input

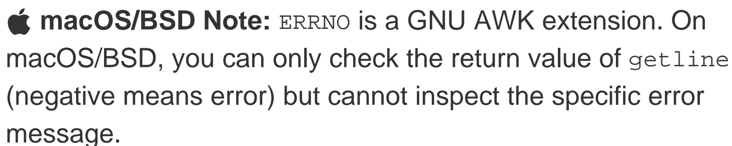
```
# Skip malformed records
NF < 3 {
    printf "WARN: line %d has %d fields, expected 3+\n", NR,
NF > "/dev/stderr"
    next
}
```

```
# Validate numeric fields
$3 !~ /^-?[0-9]+(\.[0-9]+)?$/ {
    printf "WARN: line %d field 3 is not numeric: %s\n", NR,
$3 > "/dev/stderr"
    next
}
```

ERRNO

GNU AWK sets the `ERRNO` variable when I/O operations fail. It contains a human-readable error string.

```
{
    if ((getline line < $1) < 0)
        print "Cannot open", $1, "-", ERRNO > "/dev/stderr"
    close($1)
}
```



> **macOS/BSD Note:** `ERRNO` is a GNU AWK extension. On macOS/BSD, you can only check the return value of `getline` (negative means error) but cannot inspect the specific error message.

Defensive Patterns

```
# Ensure BEGIN sets up required state
BEGIN {
    if (ARGC < 2) {
        print "Usage: script.awk <datafile>" > "/dev/stderr"
        exit 1
    }
    FS = ","
}

# Bail out on too many errors
{
    if (NF != expected_fields) {
        errors++
```

```
        if (errors > 100) {
            print "Too many format errors, aborting" >
"/dev/stderr"
            exit 2
        }
        next
    }
}

# Ensure files are always closed
END {
    close("output.txt")
}
```

Chapter 13: Practical Recipes

Text Processing

```
# Print lines matching a pattern
$ awk '/ERROR/' server.log

# Print specific columns from CSV
$ awk -F, '{ print $1, $3 }' data.csv

# Remove blank lines
$ awk 'NF > 0' file.txt

# Remove duplicate lines (preserving order)
$ awk '!seen[$0]++' file.txt

# Print lines between two markers (exclusive)
$ awk '/START/{f=1; next} /END/{f=0} f' file.txt

# Number non-blank lines
$ awk 'NF { printf "%4d\t%s\n", ++n, $0; next } { print }'
file.txt

# Print the nth line
$ awk 'NR==10' file.txt

# Print the last line
$ awk 'END { print }' file.txt

# Print lines longer than 80 characters
$ awk 'length > 80' file.txt

# Reverse field order
$ awk '{ for (i=NF; i>0; i--) printf "%s%s", $i, (i>1 ? OFS :
ORS) }' file.txt

# Head (first N lines)
$ awk 'NR<=10' file.txt

# Tail (last N lines) — buffered
```

```
$ awk '{ buf[NR%10]=$0 } END { for (i=NR-9; i<=NR; i++) print
buf[i%10] }' file.txt
```

Field Manipulation

```
# Swap two columns
$ awk '{ tmp=$1; $1=$2; $2=tmp; print }' file.txt

# Add a new column
$ awk '{ print $0, NR }' file.txt

# Remove a column (e.g., second) — clean version
$ awk '{
    for (i=1; i<=NF; i++)
        if (i != 2)
            printf "%s%s", $i, (i<NF ? OFS : ORS)
}' file.txt

# Sum a column
$ awk '{ sum += $3 } END { print sum }' data.txt

# Running total
$ awk '{ sum += $3; print $0, sum }' data.txt

# Average a column
$ awk '{ sum += $3; n++ } END { if (n > 0) print sum/n }'
data.txt

# Max of a column
$ awk 'NR==1 || $3>max { max=$3 } END { print max }' data.txt

# Min of a column
$ awk 'NR==1 { min=$3 } $3<min { min=$3 } END { print min }'
data.txt

# Standard deviation
$ awk '{
    sum += $1; sumsq += $1*$1; n++
} END {
    mean = sum/n
    printf "mean=%.2f stddev=%.2f\n", mean, sqrt(sumsq/n -
mean*mean)
```

```
}' data.txt
```

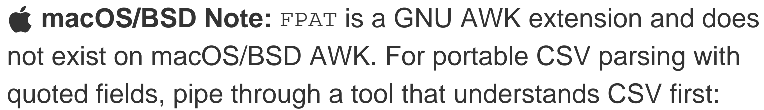
CSV Handling

AWK is not a native CSV parser — quoted fields with embedded commas or newlines will break naive splitting. For simple CSV (no embedded commas or newlines):

```
# Basic CSV field extraction
$ awk -F, '{ print $1, $3 }' data.csv

# Skip header
$ awk -F, 'NR > 1 { print $1, $3 }' data.csv

# Filter CSV rows
$ awk -F, '$3 > 1000' sales.csv

# CSV with header — access fields by name
$ awk -F, '
NR == 1 {
    for (i = 1; i <= NF; i++)
        col[$i] = i
    next
}
{
    print $(col["name"]), $(col["revenue"])
}
' data.csv
```

For proper CSV with quoting, use `FPAT` to define what a field *looks like* rather than what separates fields:

```
BEGIN {
    FPAT = "([^,]*)|(\"[^\"]*\")"
    OFS = ","
}
{
    # $1, $2, etc. now correctly handle quoted fields
    print $1, $3
}
```



macOS/BSD Note: `FPAT` is a GNU AWK extension and does not exist on macOS/BSD AWK. For portable CSV parsing with quoted fields, pipe through a tool that understands CSV first:

```
# Convert proper CSV to tab-separated, then process with awk
python3 -c "
import csv, sys
for row in csv.reader(sys.stdin):
    print('\t'.join(row))
" < data.csv | awk -F'\t' '{ print $1, $3 }'
```

Multi-Line Records

Many real-world formats use blank lines, delimiters, or fixed-line groups to separate records. AWK handles all of these.

Paragraph Mode

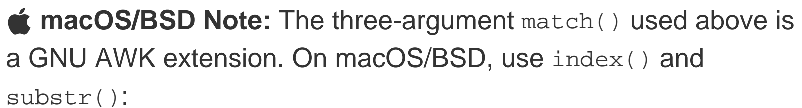
Setting `RS=""` treats blocks of text separated by one or more blank lines as single records. Within each block, `FS` splits fields as normal. This is the simplest way to handle multi-line records.

```
# Parse address blocks separated by blank lines:
# Alice Smith
# 42 Oak Street
# Belfast BT1 1AA
#
# Bob Jones
# 17 Elm Road
# Dublin D01 F5P2

BEGIN { RS = ""; FS = "\n" }
{
    name = $1
    street = $2
    city = $3
    printf "%-20s %s, %s\n", name, street, city
}
```

Mail/HTTP Headers

Headers are paragraph-mode records where each line is a key-value pair.

```
# Parse email headers — extract From and Subject
BEGIN { RS = ""; FS = "\n" }
{
    delete hdr
    for (i = 1; i <= NF; i++) {
        if (match($i, /^([^:]+): *(.*)/, arr))
            hdr[arr[1]] = arr[2]
    }
    if ("From" in hdr && "Subject" in hdr)
        printf "%-30s %s\n", hdr["From"], hdr["Subject"]
}
```



> **macOS/BSD Note:** The three-argument `match()` used above is a GNU AWK extension. On macOS/BSD, use `index()` and `substr()`:

```
for (i = 1; i <= NF; i++) {
    p = index($i, ":")
    if (p > 0) {
        key = substr($i, 1, p-1)
        val = substr($i, p+1)
        gsub(/^ +/, "", val)
        hdr[key] = val
    }
}
```

Fixed-Line Groups

Some formats use a fixed number of lines per record (e.g., FASTQ bioinformatics files have exactly 4 lines per record).

```
# Process FASTQ: 4 lines per record (header, sequence, +,
quality)
{
```

```
    header = $0
    getline sequence
    getline              # skip the "+" line
    getline quality

    # Count sequences longer than 100bp
    if (length(sequence) > 100) long++
}
END { print long, "sequences > 100bp" }
```

Custom Delimiters

When records are separated by a specific marker line rather than blank lines:

```
# Records separated by "---" lines
BEGIN { RS = "---\n"; FS = "\n" }
NF > 0 {
    for (i = 1; i <= NF; i++)
        print NR, i, $i
}
```

> **macOS/BSD Note:** The multi-character regex `RS` above is a GNU AWK extension. On macOS/BSD, use a state machine approach:

```
/^---$/ { process_record(); rec = ""; next }
{ rec = rec (rec ? "\n" : "") $0 }
END { if (rec) process_record() }
function process_record() {
    n = split(rec, lines, "\n")
    for (i = 1; i <= n; i++)
        print lines[i]
}
```

Slurp Mode

For truly complex multi-line parsing where record boundaries are not easily expressed as separators:

```
# Extract all function signatures from a C file
BEGIN { RS = "\0" }
{
    while (match($0, /[a-zA-Z_][a-zA-Z_0-9]*
+[a-zA-Z_][a-zA-Z_0-9]*\([^)]*\)/)) {
        print substr($0, RSTART, RLENGTH)
        $0 = substr($0, RSTART + RLENGTH)
    }
}
```

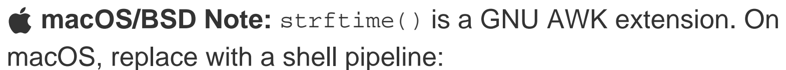
Log Analysis

```
# Count occurrences of each HTTP status code
$ awk '{ count[$9]++ } END { for (c in count) print c,
count[c] }' access.log

# Extract IPs making more than 100 requests
$ awk '{ ip[$1]++ } END { for (i in ip) if (ip[i]>100) print
ip[i], i }' access.log

# Calculate average response time (field 10)
$ awk '{ sum+=$10; n++ } END { printf "%.2fms\n", sum/n }'
access.log

# Top 10 requested URLs (with sort)
$ awk '{ url[$7]++ } END { for (u in url) print url[u], u }'
access.log | sort -rn | head -10

# Requests per hour
$ awk -F'[[ :]' '{ hour[$5]++ } END { for (h in hour) print h,
hour[h] }' access.log | sort

# Error rate by endpoint
$ awk '{
    endpoint[$7]++
    if ($9 >= 400) errors[$7]++
} END {
    for (ep in endpoint)
        printf "%-40s %6d requests  %5.1f%% errors\n",
            ep, endpoint[ep],
            (ep in errors ? errors[ep]/endpoint[ep]*100 : 0)
```

```
}' access.log | sort -t'%' -k1 -rn
```

System Administration

```
# List users with a real shell
$ awk -F: '$NF !~ /(nologin|false)$/ { print $1 }' /etc/passwd

# Processes using more than 1GB RSS (ps aux output)
$ ps aux | awk 'NR>1 && $6 > 1048576 { print $11, $6/1024 "MB"
}'

# Disk usage summary — largest first
$ df -h | awk 'NR>1 { print $5, $6 }' | sort -rn

# Monitor a log in real time
$ tail -f app.log | awk '/ERROR/ { print strftime("%H:%M:%S"),
$0; fflush() }'

# Count lines in each file (like wc -l but with totals)
$ find . -name '*.go' -exec awk 'END { print FILENAME, FNR }'
{} + |
  awk '{ total += $2; print } END { print "TOTAL", total }'
```



> **macOS/BSD Note:** `strftime()` is a GNU AWK extension. On macOS, replace with a shell pipeline:

```
tail -f app.log | awk '/ERROR/ { fflush() }' | while IFS= read
-r line; do printf '%s %s\n' "$(date +%H:%M:%S)" "$line"; done
```

Data Transformation

```
# Convert TSV to CSV
$ awk 'BEGIN { FS="\t"; OFS="," } { $1=$1; print }' data.tsv

# JSON-ish output (for simple flat records)
$ awk -F, 'NR>1 { printf "{\"name\":\"%s\",\"value\":%s}\n",
$1, $2 }' data.csv

# Transpose rows and columns
$ awk '{
```

```
    for (i = 1; i <= NF; i++)
        a[i, NR] = $i
    if (NF > max) max = NF
}
END {
    for (i = 1; i <= max; i++) {
        for (j = 1; j <= NR; j++)
            printf "%s%s", a[i, j], (j < NR ? OFS : "")
        print ""
    }
}' data.txt

# Group records — blank line between groups
$ awk -F, '
    prev && $1 != prev { print "" }
    { print; prev = $1 }
' sorted_data.csv

# Histogram of word lengths
$ awk '{ for (i=1; i<=NF; i++) len[length($i)]++ }
  END { for (l in len) printf "%3d: %s (%d)\n", l,
substr("####################",1,len[l]), len[l] }' file.txt
```

Joining Data

```
# Simple lookup join (like a hash join)
$ awk -F, 'NR==FNR { lookup[$1]=$2; next } $3 in lookup {
print $0, lookup[$3] }' \
    lookup.csv data.csv

# Left join — include unmatched rows
$ awk -F, 'NR==FNR { lookup[$1]=$2; next }
    { print $0, ($3 in lookup) ? lookup[$3] : "N/A" }' \
    lookup.csv data.csv

# Anti-join — rows in file2 with no match in file1
$ awk -F, 'NR==FNR { seen[$1]; next } !($1 in seen)' file1.csv
file2.csv
```

The `NR==FNR` idiom is essential: `FNR` resets per file while `NR` does not, so `NR==FNR` is true only while processing the first file.

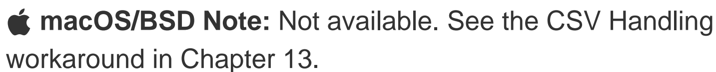
Chapter 14: GNU AWK Extensions

These features are specific to GNU AWK. Each section includes a portability note.

FPAT — Field Defined by Content

Instead of defining what *separates* fields, define what *constitutes* a field.

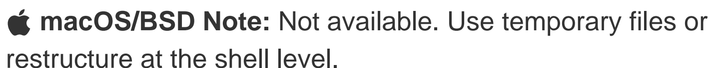
```
BEGIN { FPAT = "([^,]*)|(\"[^\"]*\")" }
{ print $2 }
```



> **macOS/BSD Note:** Not available. See the CSV Handling workaround in Chapter 13.

Coprocess (Two-Way Pipe)

```
{
    print "input" |& "/usr/bin/program"
    "/usr/bin/program" |& getline result
    print result
}
```



> **macOS/BSD Note:** Not available. Use temporary files or restructure at the shell level.

Network I/O

```
BEGIN {
    server = "/inet/tcp/0/example.com/80"
    print "GET / HTTP/1.0\r\nHost: example.com\r\n" |& server
    while ((server |& getline line) > 0)
        print line
    close(server)
}
```

> 🍎 **macOS/BSD Note:** Not available. Use `curl` or `nc` in a shell pipeline instead.

@include and @load

```
@include "library.awk"
@load "filefuncs"
```

> 🍎 **macOS/BSD Note:** Not available. Use multiple `-f` flags to load several AWK source files:
>
> ```
> awk -f library.awk -f main.awk data.txt
> ```

Typed Regex (gawk 5+)

A regex literal that can be stored in a variable and passed to functions without the overhead of runtime compilation.

```
BEGIN { pattern = @/^ERROR/ }
$0 ~ pattern { print }
```

> 🍎 **macOS/BSD Note:** Not available. Use a string variable and the `~` operator: `pattern = "^ERROR"`. This works identically but recompiles the regex each time it is evaluated.

BEGINFILE and ENDFILE

```
BEGINFILE {
    if (ERRNO) {
        print FILENAME ": " ERRNO > "/dev/stderr"
        nextfile
    }
}
ENDFILE {
    print FILENAME, FNR, "records"
}
```

macOS/BSD Note: Not available. Use the `FNR == 1` idiom to detect file boundaries:

```
FNR == 1 && NR > 1 { print prev_file, prev_fnr, "records" }
FNR == 1 { prev_file = FILENAME }
{ prev_fnr = FNR }
END { print FILENAME, FNR, "records" }
```

length(array)

GNU AWK allows `length()` to return the number of elements in an array.

```
END { print "Unique keys:", length(counts) }
```

macOS/BSD Note: Not available. Count manually:

```
END { n=0; for (k in counts) n++; print "Unique keys:", n }
```

RT — The Matched Record Terminator

When `RS` is a regex, `RT` contains the actual text that matched `RS` for the current record.

```
BEGIN { RS = "[.!?]" }
{ print NR, $0, "[" RT "]" }
```

macOS/BSD Note: `RT` does not exist on macOS/BSD AWK, and regex `RS` is also unsupported.

Chapter 15: Debugging and Profiling

--lint

The `--lint` flag warns about constructs that are dubious, non-portable, or likely bugs.

```
$ awk --lint -f script.awk data.txt
```

Common warnings include: use of uninitialised variables in comparisons, regex constants used in boolean context, and escape sequences that may not be portable.

Use `--lint=fatal` to turn warnings into errors — useful in CI pipelines.

--dump-variables

Writes all global variables and their final values to a file after execution.

```
$ awk --dump-variables -f script.awk data.txt
$ cat awkvars.out
```

This is invaluable for debugging: if a variable has an unexpected value, `--dump-variables` tells you every variable's state at program exit. You can specify the output file with `--dump-variables=file`.

--profile

Generates a "pretty-printed" version of your AWK program with execution counts — how many times each rule and statement ran.

```
$ awk --profile -f script.awk data.txt
$ cat awkprof.out
```

The output shows your program reformatted with annotations like:

```
# Rule(s)

     1000  $3 > 100 {      # 342
              342      print $1, $3
         }
```

This means the pattern `$3 > 100` was tested 1000 times and matched 342 times. Useful for finding rules that never match (dead code) or match far more than expected (performance bottleneck).

Specify the output file with `--profile=file`.

dgawk — The Debugger

GNU AWK includes an interactive debugger, invoked with `--debug` or `-D`.

```
$ awk --debug -f script.awk data.txt
```

The debugger supports:

Command	Action
break / b	Set breakpoint at line or function
run / r	Start execution
next / n	Step over
step / s	Step into function
continue / c	Resume execution
print / p	Print variable value
watch	Break when variable changes
backtrace / bt	Show call stack
list / l	Show source
quit / q	Exit debugger

```
gawk> b 15
Breakpoint 1 set at line 15
gawk> r
Stopped at line 15
gawk> p $0
$0 = "alice,engineering,95000"
gawk> p revenue["alice"]
revenue["alice"] = 95000
gawk> c
```

--sandbox

The `--sandbox` flag disables all commands and features that could affect the outside world:

- `system()` calls are blocked
- Output redirection (`>`, `>>`) is blocked
- Input redirection (`<`, `getline < file`) is blocked
- Pipes (`|`, `|&`) are blocked
- The `ARGV` array is read-only

```
$ awk --sandbox '{ system("rm -rf /") }' data.txt
awk: cmd. line:1: fatal: `system' function not allowed in
sandbox mode
```

This is essential when running AWK on untrusted input — for instance, processing user-submitted data where field values might be crafted to exploit `system()` calls or redirect output to overwrite files. If your AWK script only needs to read from stdin and write to stdout, `--sandbox` eliminates an entire class of risk at no cost.

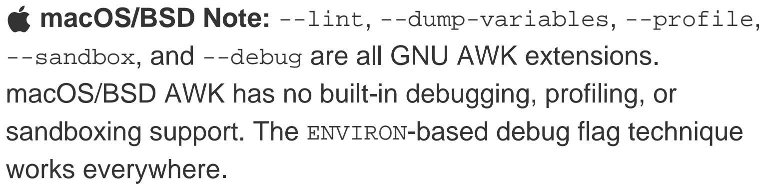
> **macOS/BSD Note:** `--sandbox` is a GNU AWK extension. macOS/BSD AWK has no equivalent — you must audit your script manually to ensure it does not use `system()` or I/O redirection if security is a concern.

Quick Debugging Without the Debugger

When you do not need the full debugger, these techniques help:

```
# Print to stderr so debug output doesn't mix with data
{ print "DEBUG: NR=" NR " NF=" NF " $0=" $0 > "/dev/stderr" }

# Conditional debug output controlled by a variable
BEGIN { DEBUG = ENVIRON["AWK_DEBUG"] }
DEBUG { print "DEBUG:", NR, $0 > "/dev/stderr" }
```

```
$ AWK_DEBUG=1 awk -f script.awk data.txt   # debug on
$ awk -f script.awk data.txt                 # debug off
```



macOS/BSD Note: `--lint`, `--dump-variables`, `--profile`, `--sandbox`, and `--debug` are all GNU AWK extensions. macOS/BSD AWK has no built-in debugging, profiling, or sandboxing support. The `ENVIRON`-based debug flag technique works everywhere.

Chapter 16: Idioms and Patterns

The NR==FNR Idiom

Process the first file into an array, then use it while processing subsequent files.

```
NR==FNR { data[$1]=$2; next }
{ print $0, ($1 in data) ? data[$1] : "N/A" }
```

Counting and Aggregation

```
# Frequency count
{ count[$1]++ }

# Group-by sum
{ sum[$1] += $3 }

# Unique values
{ seen[$2] = 1 }
END { for (v in seen) print v }

# Top-N (accumulate, sort in END)
END {
    for (k in count)
        print count[k], k | "sort -rn | head -10"
}
```

State Machine

```
/^BEGIN BLOCK/ { capture = 1; next }
/^END BLOCK/   { capture = 0; next }
capture        { buffer = buffer $0 "\n" }
```

Accumulate and Print

```
{
    lines[NR] = $0
}
END {
    for (i = NR; i >= 1; i--)
        print lines[i]
}
```

Multi-File Awareness

```
FNR == 1 {
    if (filenum > 0)
        process_previous_file()
    filenum++
}
{ ... }
END { process_previous_file() }
```

The 1 Idiom

A bare `1` at the end of a program is shorthand for `{ print }` — it is a pattern that is always true, and the default action is to print `$0`.

```
# These are equivalent:
{ gsub(/old/, "new"); print }
{ gsub(/old/, "new") } 1
```

This idiom is common in one-liners. It separates the transformation from the output, making it clear that you want to print every record regardless of the transformation.

Chapter 17: Locale and Encoding

How Locale Affects AWK

AWK's behaviour is influenced by the `LC_*` environment variables, particularly `LC_COLLATE` and `LC_CTYPE`. This affects:

- **Character classes**: `[[:alpha:]]`, `[[:upper:]]`, `[[:lower:]]`, etc.
- **Case conversion**: `tolower()` and `toupper()`
- **Sort order in string comparisons**: `"ä" < "z"` may vary
- **Range expressions in brackets**: `[a-z]` may include or exclude accented characters

The Problem in Practice

In the `C` or `POSIX` locale, `[a-z]` means exactly the 26 lowercase ASCII letters. In a UTF-8 locale (e.g., `en_US.UTF-8`), `[a-z]` may include accented characters like `ä`, `é`, `ñ`, and so on, depending on the collation order. This produces real bugs:

```
# In C locale: matches only ASCII lowercase
$ LC_ALL=C awk '/^[a-z]+$/' file.txt

# In UTF-8 locale: may match accented characters too
$ LC_ALL=en_US.UTF-8 awk '/^[a-z]+$/' file.txt
```

Scenario 1: The Log Parser That Breaks in Docker

You write a log parser on your Mac that extracts fields matching `[a-zA-Z_]+`. It works locally. In your Docker container (which runs `debian:slim` with `LANG=C.UTF-8`), the same script silently matches accented characters from user-submitted data, producing garbled keys in your aggregation arrays. The fix:

```
# Force C locale in the script or at invocation
$ LC_ALL=C awk -f parser.awk access.log
```

Or inside the AWK program itself:

```
BEGIN {
    # Document the assumption
    if (ENVIRON["LC_ALL"] != "C" && ENVIRON["LANG"] !~ /^C/)
        print "WARNING: script expects C locale" >
"/dev/stderr"
}
```

Scenario 2: Case-Insensitive Matching with International Names

You have a user database with names like "Ólafur" and "François". A naive case-insensitive match fails:

```
# In C locale: tolower("Ó") returns "Ó" unchanged
# In UTF-8 locale: tolower("Ó") returns "ó"
tolower($1) ~ /olafur/  # only works in UTF-8 locale
```

If you need to process international text, ensure the right locale is set:

```
$ LC_ALL=en_US.UTF-8 awk '{ if (tolower($1) ~ /ólafur/) print
}' users.txt
```

Scenario 3: Sorting Surprise

String comparison order changes with locale. In the `C` locale, uppercase letters sort before lowercase (`"Z" < "a"` is true). In many UTF-8 locales, case-insensitive collation is used, so `"a" < "Z"` may be true.

```
# This comparison produces different results depending on
locale:
$1 < "M"  # In C locale: true for A-L and all symbols/digits
           # In en_US.UTF-8: true for a-l and A-L (case-folded
collation)
```

Recommendations

For predictable ASCII-only matching, set `LC_ALL=C` at invocation. This is the safest default for scripts that process structured data (logs, CSV, config files):

```
$ LC_ALL=C awk -f script.awk data.txt
```

Use POSIX classes when the intent is semantic. `[[:digit:]]` always means digits 0-9 regardless of locale. `[[:alpha:]]` means "alphabetic in this locale" which is correct when processing human-readable text.

For international text processing, ensure `LC_ALL` is set to an appropriate UTF-8 locale and test with representative data that includes non-ASCII characters.

For maximum portability, avoid `[a-z]` and `[A-Z]` ranges entirely in scripts that may run across different systems and locales. Use `[[:lower:]]`, `[[:upper:]]`, or explicit character lists.

> **macOS/BSD Note:** Locale handling is consistent between GNU AWK and macOS/BSD AWK at the POSIX level — both respect `LC_*` variables. The differences are in edge cases of Unicode handling, where gawk tends to be more thoroughly tested. macOS defaults to `en_US.UTF-8` while many Linux distributions default to `C.UTF-8` or `POSIX` in containers. This mismatch is itself a source of cross-platform bugs.

Chapter 18: Gotchas and Edge Cases

Uninitialised variables are `""` and `0`. This is usually convenient, but watch for silent bugs when testing membership in arrays — `if (arr[x])` will create a key `x` with value `""`. Use `if (x in arr)` instead.

String comparison vs numeric comparison. If both operands look like numbers, AWK compares numerically. If either is clearly a string, it compares lexicographically. Force numeric comparison with `+0`; force string with `""`. The rules are subtle — see Chapter 5 for details.

`print` vs `printf`. `print` adds `ORS` automatically. `printf` does not. Forgetting `\n` in `printf` is a common mistake, and it leads to output that looks concatenated.

OFMT and CONVFMT silently lose precision. Both default to `"%.6g"`, which gives six significant digits. Large integers (IDs, timestamps), high-precision decimals, and numbers used as array keys are all affected. A 13-digit ID becomes `1.23457e+12`. See Chapter 5 for details and fixes.

Pipe handling. The string used with `|` is the pipe handle. `print | "sort"` and `print | "sort -r"` are two different pipes. If you change the command string, close the old one first.

`getline` side effects. Different forms of `getline` have different side effects on `$0`, fields, `NR`, and `FNR`. See the table in Chapter 8. A bare `getline` overwrites `$0` and recomputes all fields — use `getline var` to avoid this.

Array order is undefined. `for (k in arr)` traverses in unspecified order. Sort explicitly or use GNU AWK's `PROCINFO["sorted_in"]`.

RS as regex vs single character. In GNU AWK, `RS` is a full regex. On macOS/BSD AWK, only the first character of `RS` is used. This is the most common silent portability bug — your script appears to work but silently misparses records.

Modifying `$0` rebuilds fields; modifying fields rebuilds `$0`. Be mindful of this coupling when doing both in the same rule. Assigning to any `$n` triggers a rebuild of `$0` using the current `OFS`. If you set `OFS` to something different from `FS`, a mere field assignment changes your entire record format.

Hex and octal in input. GNU AWK recognises `0x1F` and `077` as hex and octal in input when using the default (non-POSIX) mode. macOS/BSD AWK does not. Force decimal interpretation with `+0` if needed, or use `--posix` in GNU AWK.

Backslash in replacement strings. In `sub()` and `gsub()`, `\` in the replacement string is an escape character. To get a literal backslash, use `\\\\` (double-escaped through AWK's string parsing and then the replacement engine). GNU AWK and macOS/BSD AWK handle edge cases here slightly differently — always test if your replacement strings contain backslashes.

Concatenation has no operator. Adjacent values concatenate: `a b` is `a` concatenated with `b`. This means `print $1 $2` prints the concatenation, while `print $1, $2` prints with `OFS` between them. A missing comma is a silent bug.

`close()` is essential in loops. If you pipe to different commands inside a loop, or read from different files, you must `close()` each one explicitly. AWK does not auto-close on reassignment — the pipe or file stays open until you close it or the program ends, and most systems limit the number of open file descriptors.

Empty `FS` splits characters. Setting `FS = ""` causes each character to become a separate field. This is specified in POSIX but surprises people who expected it to mean "default whitespace."

Implementation Limits

AWK is not infinite. These limits vary by implementation but can bite you in production with large or unusual data.

Limit	GNU AWK	macOS/BSD AWK	Notes
Max fields per record	Effectively unlimited (memory-bound)	~32,767 in some versions	Fails silently or crashes with very wide CSV
Max record length	Effectively unlimited (memory-bound)	~3,000-10,000 bytes in some versions	Long lines may be truncated
Max open files/pipes	System ulimit (typically 1024)	System ulimit	Hit this in loops that open files per-key without close()
Max regex complexity	Large but bounded	Smaller	Deeply nested alternations may fail
Numeric precision	IEEE 754 double (53 bits / ~15 significant digits)	IEEE 754 double	Same on both — no arbitrary precision
Array size	Memory-bound	Memory-bound	But pathological hash collisions can degrade performance

If you hit file descriptor limits, the symptom is usually a cryptic error on `print > filename` or `| "command"`. The fix is always explicit `close()` calls.

If you process files with hundreds of columns (wide CSV exports, genomic data), test on both platforms. GNU AWK handles this gracefully; older BSD AWK may not.

For integer arithmetic beyond 2^53 (~9 × 10^15), AWK will silently lose precision. If you need exact large integers, use `printf "%d"` for output and avoid arithmetic on them — treat them as strings.

Chapter 19: AWK vs Other Tools

The UNIX Text Processing Toolkit

<table>
<tr><th>Task</th><th>AWK</th><th>Simpler Alternative</th></tr>
<tr><td>Extract a column</td><td>awk '{print $2}'</td><td>cut -f2</td></tr>
<tr><td>Find matching lines</td><td>awk '/pat/'</td><td>grep pat</td></tr>
<tr><td>Replace text</td><td>awk '{gsub(/a/,"b")}1'</td><td>sed 's/a/b/g'</td></tr>
<tr><td>Sum numbers</td><td>awk '{s+=$1}END{print s}'</td><td>paste -sd+ | bc</td></tr>
<tr><td>Count matching lines</td><td>awk '/pat/{n++}END{print n}'</td><td>grep -c pat</td></tr>
<tr><td>Sort and unique</td><td>awk '!a[$0]++' (order preserved)</td><td>sort -u (sorted)</td></tr>
</table>

When the job fits a single tool (`grep` for matching, `cut` for column extraction, `sed` for substitution), use that tool. AWK's strength is combining these operations in a single pass without a pipeline.

AWK vs Modern Alternatives

Task	AWK	Modern Tool
Structured CSV/TSV	awk -F, (fragile with quoting)	mlr (Miller) — native CSV, JSON, TSV support
JSON processing	Not practical	jq — purpose-built for JSON
Complex CSV queries	FPAT + manual logic	csvq — SQL-like queries on CSV
Data analysis / statistics	Manual aggregation	datamash — column stats in one command

Multi-format ETL	Possible but verbose	mlr — chain verbs, convert between formats

Miller (`mlr`) deserves specific mention. It handles what AWK handles, plus format-awareness (CSV quoting, JSON nesting, header-based field references). If you find yourself writing defensive CSV parsing or manual JSON construction in AWK, Miller is probably the right tool.

`jq` is to JSON what AWK is to line-oriented text. Do not try to parse JSON with AWK — it is technically possible but you will regret it.

`datamash` is useful when you need column statistics (mean, median, mode, stddev, percentiles) without writing the aggregation yourself. AWK can do all of this manually, but `datamash` does it in one command.

When AWK Wins

AWK is the right choice when the data is line-oriented with predictable delimiters, the processing involves field extraction, filtering, aggregation, or multi-file joins, and the logic is complex enough that a `grep | sed | cut` pipeline becomes unreadable but simple enough that a Python script is overkill.

AWK's sweet spot is the space between "this pipeline is getting unwieldy" and "I need a real programming language." It processes million-line files with negligible overhead, runs everywhere, and requires no installation.

When to Reach Elsewhere

- **Deeply nested or hierarchical data** (JSON, XML, YAML) — use `jq`, `xmlstarlet`, or `yq`.
- **Complex CSV with quoting, embedded newlines, or multi-line fields** — use `mlr`, `csvtool`, or Python's `csv` module.

- **Binary data** — AWK is a text tool.
- **Anything requiring libraries** (HTTP clients, database drivers, encryption) — use a general-purpose language.
- **Large integers or exact arithmetic** — AWK uses IEEE 754 doubles. Use `bc` or a language with arbitrary precision.

Chapter 20: Cross-Platform Compatibility

This chapter is a quick-reference for writing AWK that works on both GNU AWK (Linux) and macOS/BSD AWK.

What Is macOS/BSD AWK?

macOS ships `/usr/bin/awk`, which is based on Brian Kernighan's "one true awk" — a capable POSIX-compliant implementation, but without any GNU extensions. It has not been significantly updated in years. FreeBSD and OpenBSD ship similar POSIX-focused implementations.

If you need GNU AWK on macOS:

```
$ brew install gawk
$ gawk --version
```

Feature Compatibility Matrix

Feature	GNU AWK	macOS/BSD AWK	Portable Alternative
gensub()	.	.	match() + substr()
FPAT	.	.	Pre-process CSV externally
strftime() / mktime()	.	.	Shell out to date
patsplit()	.	.	split() + loops
match(s, r, arr) (3rd arg)	.	.	Multiple match() calls
length(array)	.	.	Manual count loop

Two-way pipe |&	.	.	Temp files or shell pipeline
Regex RS	.	.	Single-char RS only
PROCINFO	.	.	Pipe to sort
ERRNO	.	.	Check getline return value
RT	.	.	Not available
BEGINFILE / ENDFILE	.	.	FNR == 1 idiom
@include / @load	.	.	Multiple -f flags
Typed regex @/pat/	.	.	String variable with ~
arr[i][j] (true multidim)	.	.	arr[i, j] with SUBSEP
Network I/O /inet/	.	.	curl / nc pipeline
-i / -l flags	.	.	No equivalent
--lint / --profile	.	.	No equivalent
--debug (debugger)	.	.	No equivalent
--sandbox	.	.	Manual audit
delete arr (whole array)	.	..	Loop: for (k in arr) delete arr[k]
\x hex escapes	.	.	printf with octal
ENVIRON	.	.	—
FNR	.	.	—
tolower() / toupper()	.	.	—

nextfile	.	.	—
Regex FS	.	.	—
Paragraph mode RS=""	.	.	—
User-defined functions	.	.	—
getline (all forms)	.	.	—
SUBSEP multi-dim arrays	.	.	—
POSIX character classes	.	.	—

Strategy: Write for GNU, Know the Boundaries

If your AWK scripts run exclusively on Linux, use GNU AWK freely. If they might run on macOS (common in teams with mixed workstations, CI environments, or open-source projects), follow this approach:

1. **Default to POSIX features.** The core language — fields, patterns, arrays, `sub`/`gsub`, `match`, `split`, `printf` — is identical across implementations and covers the vast majority of real-world use.
2. **Reach for GNU extensions deliberately.** `gensub`, `FPAT`, and `strftime` solve real problems that are painful without them. Use them when they earn their place, and document the dependency.
3. **Test on both.** The easiest way to catch portability issues is to run your script under both implementations. If you have Homebrew on macOS, you have access to both:

```
# macOS system AWK
/usr/bin/awk -f script.awk data.txt

# GNU AWK via Homebrew
gawk -f script.awk data.txt
```

4. **Use a shebang that matches your intent.** If the script requires GNU AWK, say so:

```
#!/usr/bin/env gawk -f
```

   If it is portable:

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

5. **Use `--lint` during development.** It catches many portability issues and dubious constructs that would silently fail on other implementations.

Appendix A: One-Liner Cheat Sheet

```
# Print line count
awk 'END{print NR}' file

# Print word count
awk '{w+=NF}END{print w}' file

# Print character count (including newlines)
awk '{c+=length+1}END{print c}' file

# Print unique lines (in order)
awk '!a[$0]++' file

# Print duplicate lines only
awk 'a[$0]++' file

# Print every 5th line
awk 'NR%5==0' file

# Print lines 10-20
awk 'NR>=10&&NR<=20' file

# Delete trailing whitespace
awk '{sub(/[[:space:]]+$/,"")}1' file

# Delete leading whitespace
awk '{sub(/^[[:space:]]+/,"")}1' file

# Squeeze multiple spaces to one
awk '{gsub(/  +/," ")}1' file

# Add line numbers
awk '{print NR": "$0}' file

# Sum all numbers in a file
awk '{for(i=1;i<=NF;i++)s+=$i}END{print s}' file

# Collapse multiple blank lines to one
awk 'NF{blank=0}!NF{blank++}blank<=1' file
```

```
# Print the longest line
awk '{if(length>max){max=length;line=$0}}END{print line}' file

# Print lines that appear exactly once
awk '{a[$0]++}END{for(l in a)if(a[l]==1)print l}' file

# Interleave two files line by line
awk '{print; if((getline line < "file2.txt")>0) print line}'
file1.txt

# Cross-tabulation / pivot (GNU AWK — uses true multidim
arrays)
awk -F, '{data[$1][$2]+=$3}END{for(r in data)for(c in
data[r])print r,c,data[r][c]}' file
```

> **macOS/BSD Note:** The cross-tabulation one-liner above uses GNU AWK's true multi-dimensional arrays. The portable version:
>
> ```
> awk -F, '{data[$1","$2]+=$3}END{for(k in
> data){split(k,a,",");print a[1],a[2],data[k]}}' file
> ```
>
> All other one-liners in this appendix are fully portable.

Appendix B: Quick Reference Card

Invocation

```
awk [-F fs] [-v var=val] 'program' files...
awk [-F fs] [-v var=val] -f progfile files...
```

Program Structure

```
BEGIN      { startup code }
pattern    { per-record code }
END        { cleanup code }
```

Special Patterns

```
BEGIN          END
/regex/        expression
pat1, pat2     (range)
BEGINFILE      ENDFILE        (GNU AWK only)
```

Record/Field Variables

```
$0  NR  FNR  NF  FS  RS  OFS  ORS  FILENAME
ARGC  ARGV  ENVIRON  SUBSEP  RSTART  RLENGTH
PROCINFO  ERRNO  RT  (GNU AWK only)
OFMT  CONVFMT  (default "%.6g" — watch precision)
```

String Functions

```
Portable:  length  substr  index  split  sub  gsub  match
sprintf
           tolower  toupper
GNU only:  gensub  patsplit  strftime  mktime  match(s,r,arr)
```

Math Functions

```
int   sqrt   exp   log   sin   cos   atan2   rand   srand
```

I/O

```
print   printf   getline   close   system   fflush
> file    >> file    | command    |& command (GNU AWK only)
/dev/stderr   /dev/stdin   /dev/stdout
```

Getline Forms

```
Form                          Sets $0?  Sets NR/FNR?
.........................     ........  ............
getline                       Yes       Yes
getline var                   No        Yes
getline < file                Yes       No
getline var < file            No        No
cmd | getline                 Yes       No
cmd | getline var             No        No

Return: 1 = success, 0 = EOF, -1 = error
```

Regex (ERE)

```
.   ^   $   *   +   ?   |   ()   []   {n,m}
POSIX: [[:alpha:]]  [[:digit:]]  [[:space:]]  [[:upper:]]
[[:lower:]]
       [[:alnum:]]  [[:punct:]]  [[:blank:]]  [[:xdigit:]]
[[:print:]]
NOT supported: \d  \w  \s  \b  (use POSIX classes instead)
```

Operators (precedence, high to low)

```
()  $            grouping, field
^                exponent
! ++ --          unary
* / %            multiplicative
+ -              additive
(concatenation) string join
< <= > >= == != ~ !~   comparison
in               array membership
&&               logical and
||               logical or
?:               ternary
= += -= ...      assignment
```

Debugging (GNU AWK only)

```
--lint           warn about dubious constructs
--lint=fatal     treat lint warnings as errors
--dump-variables  dump all variables on exit
--profile        generate execution profile
--debug / -D     interactive debugger
--sandbox        disable system(), I/O redirection, pipes
```

Appendix C: Task Index

A quick-reference for finding what you need by what you are trying to do.

"How do I..."

Task	Where to look
...read a CSV file?	Ch 13, CSV Handling
...handle quoted CSV fields?	Ch 13, CSV Handling (FPAT); Ch 14, FPAT
...join two files?	Ch 13, Joining Data
...do a left join / anti-join?	Ch 13, Joining Data
...process multi-line records?	Ch 13, Multi-Line Records
...parse email/HTTP headers?	Ch 13, Multi-Line Records — Mail/HTTP Headers
...remove duplicate lines?	Ch 13, Text Processing
...sum / average / min / max a column?	Ch 13, Field Manipulation
...print specific lines (nth, range, last)?	Ch 13, Text Processing
...extract text matching a regex?	Ch 8, match() + substr()
...do a regex replacement with backreferences?	Ch 8, gensub
...build a formatted string without printing it?	Ch 8, sprintf
...use a variable as a regex?	Ch 6, Regex Contexts in AWK
...make my script work on macOS too?	Ch 20, Cross-Platform Compatibility
...sort AWK output?	Ch 7, Sorting Arrays; or pipe to sort

...change the record separator?	Ch 2, Records
...read a whole file into one string?	Ch 2, Records — Slurp Mode
...pass variables into AWK?	Ch 1, Invocation (-v)
...read from a second file in BEGIN?	Ch 8, I/O Functions — getline < file
...write to stderr?	Ch 9, Writing to stderr
...use AWK with pipes?	Ch 9, Pipes
...do two-way I/O with a command?	Ch 9, Two-Way Pipes; Ch 14, Coprocess
...define my own functions?	Ch 11
...create local variables in a function?	Ch 11, Local Variables
...debug an AWK script?	Ch 15
...profile AWK for performance?	Ch 15, --profile
...run AWK safely on untrusted input?	Ch 15, --sandbox
...check for portability problems?	Ch 15, --lint

"Why is my..."

Problem	Where to look
...number wrong / rounded / in scientific notation?	Ch 5, OFMT and CONVFMT
...array key losing digits?	Ch 5, OFMT and CONVFMT
...script working on Linux but not macOS?	Ch 20; Ch 18, Gotchas (RS as regex)
...regex matching unexpected characters?	Ch 17, Locale and Encoding
...case-insensitive match failing on accented characters?	Ch 17, Scenario 2
...getline clobbering my fields?	Ch 8, Getline Side Effects Table

...comparison giving wrong results?	Ch 5, Type Coercion
...pipe output mixed up or missing?	Ch 18, Gotchas — close() in loops
...field separator not working?	Ch 2, Field Separator (single space special case)
...for-in loop printing in random order?	Ch 7, Sorting Arrays
...array membership test creating empty keys?	Ch 7 — use in, not if (arr[x])
...awk script slow on many files?	Ch 18, Implementation Limits — open file descriptors

Colophon

AWK Pocket Reference — A uRadical Book Version 1.0 · 2026

Published by uRadical · uradical.io/books

Written for developers who read man pages, not marketing pages.