PenultiAWK

User Manual

Version 0.04

A Browser-Based AWK Interpreter with Broad POSIX Compatibility

Single-file, offline, no dependencies

Table of Contents

1. Introduction

PenultiAWK is a browser-based AWK interpreter designed to be as compatible with POSIX AWK as practical within a web application. It runs entirely from a single HTML file and requires no installation, internet connection, or external dependencies. Simply open the HTML file in any modern browser to use the AWK environment.

PenultiAWK is built on a lexer, parser, AST, and tree-walking interpreter architecture. It supports a broad range of the POSIX AWK language and extends it with CSV, JSON, and sort capabilities. Because it runs inside a browser, it does not claim complete POSIX conformance.

1.1 Design Philosophy

PenultiAWK is part of the Penultimate family of offline web tools. It follows these principles:

- Single-file deployment: one HTML file contains everything
- Offline operation: no network access required
- Zero installation: runs in any modern browser
- Maintainability: clean, readable source code
- Broad POSIX compatibility: many standard AWK programs run correctly, subject to browser limitations

1.2 Key Features

- Broad POSIX AWK language support (BEGIN/END, patterns, functions, arrays, etc.)
- CodeMirror 6 script editor with syntax highlighting, bracket matching, and code folding
- RFC 4180 CSV parsing and output (v0.03)
- JSON input/output with auto-detection of JSON Lines and JSON arrays (v0.03)
- GAWK-compatible sort functions: asort(), asorti(), sort_array() (v0.04)
- Sort buttons on Input and Output panels for quick line sorting (v0.04)
- Field separator (-F) and variable assignment (-v) flags
- Save/load scripts, input data, and output

2. User Interface

The PenultiAWK interface is divided into four areas: the menu bar, the script editor, the input panel, and the output panel.

2.1 Menu Bar

The menu bar runs across the top of the window and contains:

- **File menu:** New Script, Open Script, Save Script, Load Input, Save Output
- **Run menu:** Execute (Ctrl+Enter), Clear Output, Clear All
- **Help menu:** AWK Quick Reference, About

2.2 Script Editor

The top half of the window contains the AWK script editor, powered by CodeMirror 6. It provides:

- Syntax highlighting for AWK keywords, built-in functions, variables, strings, numbers, and regex
- Line numbers and active line highlighting
- Bracket matching and auto-close brackets
- Code folding for blocks
- Find and replace (Ctrl+F)
- Undo/redo history

2.3 Script Toolbar

The toolbar above the script editor contains:

- **-F** field: set the input field separator (equivalent to awk -F)
- **-v** field: set variables before execution (format: var=val,var2=val2)
- **-csv** checkbox: enable RFC 4180 CSV field splitting
- **-json** checkbox: enable JSON input mode (auto-detects JSON Lines or JSON array)
- **Run** button: execute the script (also Ctrl+Enter)

2.4 Input and Output Panels

The bottom half of the window is split into Input (left) and Output (right) panels.

Input panel toolbar: Load (load a file), Sort (sort lines alphabetically), Clear

Output panel toolbar: Save (download output), Sort (sort lines alphabetically), Copy (copy to clipboard), Clear

2.5 Status Bar

The status bar at the bottom shows the interpreter state (Ready, Running, Completed), execution time, and input line/character counts.

3. AWK Language Reference

PenultiAWK implements broad POSIX AWK language support within the constraints of a browser-based application. This section provides a concise reference to the supported language. For general AWK coverage, consult "The AWK Programming Language" by Aho, Weinberger, and Kernighan.

3.1 Program Structure

An AWK program consists of pattern-action rules and optional function definitions:

```
BEGIN { ... }              # Run before input
pattern { action }         # Run for matching lines
END { ... }                # Run after all input
function name(args) { ... } # User-defined function
```

If a rule has a pattern but no action, the default action is { print $0 }. If a rule has no pattern, the action runs for every input line.

3.2 Built-in Variables

Variable	Description
$0	Entire current record
$1, $2, ...	Fields of current record
NR	Total records read so far
NF	Number of fields in current record
FNR	Records read from current file
FS	Input field separator (default: space)
OFS	Output field separator (default: space)
RS	Input record separator (default: newline)
ORS	Output record separator (default: newline)
SUBSEP	Subscript separator for multi-dimensional arrays
RSTART, RLENGTH	Set by match() function
FILENAME	Name of current input file
ARGC, ARGV	Argument count and array
ENVIRON	Environment variables array

3.3 String Functions

Function	Description
length(s)	String length (or array size)
substr(s, start [, len])	Substring (1-based)
index(s, target)	Position of target in s (0 if not found)

split(s, array [, fs])	Split s into array; returns count
sub(regex, repl [, target])	Replace first match (default $0)
gsub(regex, repl [, target])	Replace all matches
match(s, regex)	Find regex; sets RSTART, RLENGTH
sprintf(fmt, args...)	Format string (like printf)
tolower(s)	Convert to lowercase
toupper(s)	Convert to uppercase

3.4 Math Functions

Function	Description
sin(x), cos(x)	Trigonometric functions
atan2(y, x)	Arctangent of y/x
exp(x), log(x)	Exponential and natural log
sqrt(x)	Square root
int(x)	Truncate to integer
rand()	Random number in [0, 1)
srand([seed])	Seed random number generator

3.5 I/O Statements

Statement	Description
print [args]	Print with OFS/ORS
printf fmt, args	Formatted print (no trailing ORS)
print ... > file	Redirect to file (output panel in browser)
print ... >> file	Append to file (output panel in browser)
getline [var]	Read next input line
getline [var] < file	Read from file

Note: File I/O and pipe operations are not available in the browser. Redirected output is sent to the output panel.

3.6 Control Flow

Statement	Description
if (cond) stmt [else stmt]	Conditional
while (cond) stmt	While loop
for (init; cond; upd) stmt	C-style for loop
for (key in array) stmt	Iterate array keys

do stmt while (cond)	Do-while loop
break	Exit loop
continue	Next loop iteration
next	Skip to next input record
exit [code]	Stop processing, run END
return [expr]	Return from function

4. PenultiAWK Extensions

PenultiAWK extends POSIX AWK with CSV, JSON, and sort capabilities. These extensions are designed to feel natural within the AWK programming model.

4.1 CSV Support (v0.03)

PenultiAWK provides RFC 4180-compliant CSV parsing. This handles the common pitfalls of CSV data: quoted fields, embedded commas, embedded newlines, and escaped double-quotes.

4.1.1 CSV Input Mode (-csv flag)

Check the -csv checkbox in the toolbar to enable CSV field splitting. When active, the record splitter uses the CSV parser instead of the FS-based splitter. Fields are accessible as $1, $2, etc., and NF reflects the actual CSV field count. The -F flag is ignored in CSV mode.

Example: Given this CSV input:

```
Name,City,Age
Alice,"New York, NY",30
Bob,Chicago,25
```

With -csv enabled and this script:

```
NR > 1 { print $1, "lives in", $2, "age", $3 }
```

Output:

```
Alice lives in New York, NY age 30
Bob lives in Chicago age 25
```

Note how "New York, NY" is correctly treated as a single field despite the embedded comma.

4.1.2 CSV Functions

Function	Description
csv_split(string, array)	Parse a CSV string into array (1-based). Returns field count.
csv_print(f1, f2, ...)	Output arguments as a properly quoted CSV line plus ORS.
csv_printf(fmt, f1, ...)	Format each argument with fmt, then output as CSV line.

Example: Converting tab-separated input to CSV:

```
# Set FS to tab, use csv_print for output
BEGIN { FS = "\t" }
{ csv_print($1, $2, $3) }
```

4.2 JSON Support (v0.03)

4.2.1 JSON Input Mode (-json flag)

Check the -json checkbox to enable JSON input mode. PenultiAWK auto-detects two formats:

- **JSON Lines:** one JSON object per line. Each object becomes a record.

- **JSON Array:** a single JSON array of objects. Each element becomes a record.

In JSON mode, each object's values become fields ($1, $2, ...) in key order. The key names are stored in the JSONKEYS[] array, so JSONKEYS[1] is the name of the first field, JSONKEYS[2] the second, and so on.

Example: Given this JSON Lines input:

```
{"name":"Alice","age":30,"city":"NYC"}
{"name":"Bob","age":25,"city":"LA"}
```

With -json enabled:

```
{ print $1, "is", $2, "years old in", $3 }
```

Output:

```
Alice is 30 years old in NYC
Bob is 25 years old in LA
```

4.2.2 JSON Functions

Function	Description
json_parse(string, array)	Parse a JSON string into an associative array. Objects use their keys; arrays use 1-based indices. Returns element count.
json_stringify(array)	Serialize an associative array to a pretty-printed JSON string. Produces a JSON array if keys are sequential 1-based integers; otherwise an object.

Example: Building and outputting JSON:

```
BEGIN {
    data["name"] = "Alice"
    data["age"] = 30
    data["city"] = "New York"
    print json_stringify(data)
}
```

Output:

```
{
  "name": "Alice",
  "age": 30,
  "city": "New York"
}
```

Example: Converting CSV to JSON (use -csv flag):

```
BEGIN { ORS = "" }
NR == 1 {
    split($0, headers, ",")
    n = NF
    print "[\n"
    next
}
{
    if (NR > 2) print ",\n"
    for (i = 1; i <= n; i++) rec[headers[i]] = $i
```

```
    print json_stringify(rec)
    delete rec
}
END { print "\n]\n" }
```

4.3 Sort Functions (v0.04)

PenultiAWK provides GAWK-compatible array sort functions and an extended sort function with mode control. All sort functions re-index the result array with sequential 1-based integer keys.

4.3.1 asort(source [, dest])

Sort an array by its values. The original keys are replaced with 1-based integers. If a destination array is provided, the sorted values go there and the source array is unchanged. Returns the number of elements.

Comparison is AWK-idiomatic: numeric when both values look numeric, string (lexicographic) otherwise.

```
BEGIN {
    fruit["x"] = "cherry"
    fruit["y"] = "apple"
    fruit["z"] = "banana"
    n = asort(fruit)
    for (i = 1; i <= n; i++) print fruit[i]
}
# Output: apple, banana, cherry
```

4.3.2 asorti(source [, dest])

Sort an array by its keys. The sorted keys are stored as values with 1-based integer indices. If a destination array is provided, the source is unchanged. Returns the number of elements.

```
BEGIN {
    score["charlie"] = 85
    score["alpha"] = 92
    score["bravo"] = 78
    n = asorti(score, sorted)
    for (i = 1; i <= n; i++)
        print sorted[i], score[sorted[i]]
}
# Output: alpha 92, bravo 78, charlie 85
```

4.3.3 sort_array(array, mode)

Extended sort with a mode string controlling all four dimensions:

Mode	Description
@val_num_asc	Sort by value, numeric ascending (default)
@val_num_desc	Sort by value, numeric descending
@val_str_asc	Sort by value, string (lexicographic) ascending
@val_str_desc	Sort by value, string descending
@ind_num_asc	Sort by key, numeric ascending
@ind_num_desc	Sort by key, numeric descending
@ind_str_asc	Sort by key, string ascending
@ind_str_desc	Sort by key, string descending

The @ind modes store the sorted keys as values (like asorti). The @val modes store the sorted values (like asort).

4.3.4 Sort Buttons

The Input and Output panels each have a Sort button that performs a quick lexicographic sort of the lines in that panel. This is useful for the common UNIX pattern of sorting input before processing or sorting output after processing.

5. Examples

This section provides practical examples ranging from traditional AWK one-liners to extended programs using PenultiAWK's CSV, JSON, and sort features.

5.1 Traditional AWK One-Liners

These classic one-liners work in any POSIX AWK and in PenultiAWK. Paste the input into the Input panel and the script into the editor.

Print specific fields

```
{ print $1, $3 }
```

Print lines matching a pattern

```
/error/ { print }
```

Print line numbers

```
{ print NR, $0 }
```

Sum a column

```
{ sum += $1 }
END { print "Total:", sum }
```

Count lines

```
END { print NR, "lines" }
```

Print unique lines (like sort -u)

```
!seen[$0]++ { print }
```

Reverse field order

```
{ for (i = NF; i >= 1; i--) printf "%s ", $i; print "" }
```

Print lines longer than 80 characters

```
length($0) > 80
```

Field separator example (use -F :)

```
# With /etc/passwd-style input and -F set to :
{ print $1, $3 }  # username and UID
```

Simple report with header and footer

```
BEGIN { printf "%-15s %10s\n", "Name", "Score" }
```

```
    { printf "%-15s %10d\n", $1, $2 }
END   { print "---" }
```

5.2 Word Frequency Counter

This script extracts words from text, counts their frequency, then displays results sorted alphabetically and by frequency (descending). It demonstrates the asort, asorti, sort_array, and SUBSEP features.

```
{
    line = $0
    while (match(line, /[a-zA-Z_'-]+/)) {
        word = substr(line, RSTART, RLENGTH)
        word = tolower(word)
        count[word]++
        line = substr(line, RSTART + RLENGTH)
    }
}
END {
    # Alphabetical by word
    n = asorti(count, keys)
    printf "--- Alphabetical ---\n"
    for (i = 1; i <= n; i++)
        printf "%-20s %d\n", keys[i], count[keys[i]]

    printf "\n"

    # By frequency, descending
    # Pack count and word into composite values,
    # sort descending, then unpack.
    idx = 0
    for (w in count) {
        idx++
        freq[idx] = sprintf("%010d", count[w]) SUBSEP w
    }
    sort_array(freq, "@val_str_desc")
    printf "--- By Frequency ---\n"
    for (i = 1; i <= n; i++) {
        split(freq[i], parts, SUBSEP)
        printf "%-20s %d\n", parts[2], parts[1] + 0
    }
}
```

How it works: The main loop uses match() to extract words via regex, converts to lowercase, and tallies them in count[]. In the END block, asorti() with a destination array gives alphabetical order without destroying the counts. For frequency sorting, a composite string (zero-padded count + SUBSEP + word) enables descending sort by count. The parts[1] + 0 idiom strips leading zeros.

5.3 CSV Processing Examples

Extract columns from CSV

Enable -csv and use this script:

```
# Skip header, print name and email (columns 1 and 3)
NR > 1 { print $1, $3 }
```

CSV to tab-separated

Enable -csv:

```
BEGIN { OFS = "\t" }
{ print $1, $2, $3 }
```

Filter and re-output as CSV

Enable -csv:

```
# Keep only rows where column 3 (age) > 25
NR == 1 || $3 > 25 { csv_print($1, $2, $3) }
```

Programmatic CSV parsing (without -csv flag)

```
# Use csv_split when you need CSV parsing in specific spots
{
    n = csv_split($0, fields)
    for (i = 1; i <= n; i++)
        print i ": " fields[i]
}
```

5.4 JSON Processing Examples

Extract fields from JSON Lines

Enable -json:

```
# Input: {"name":"Alice","score":95}
#        {"name":"Bob","score":87}
{ print $1, $2 }
```

Build JSON from AWK data

```
# Convert space-separated input to JSON
BEGIN { print "[" }
{
    data["name"] = $1
    data["value"] = $2 + 0
    if (NR > 1) printf ",\n"
    printf "%s", json_stringify(data)
    delete data
}
END { print "\n]" }
```

Parse JSON within a field

```
# Parse a JSON string stored in a variable
BEGIN {
    json = "{\"x\":10,\"y\":20,\"z\":30}"
    n = json_parse(json, coords)
    print "Coordinates:", coords["x"], coords["y"], coords["z"]
}
```

5.5 Sort Examples

Sort input lines and number them

```
{ lines[NR] = $0 }
END {
    n = asort(lines)
    for (i = 1; i <= n; i++)
        printf "%3d  %s\n", i, lines[i]
}
```

Top N by a numeric field

```
# Input: name score (space-separated)
{ scores[$1] = $2 + 0 }
END {
    # Sort by value descending
    n = asorti(scores, names)
    # Rebuild with values for sorting
    for (i = 1; i <= n; i++)
        ranked[i] = sprintf("%010d", scores[names[i]]) SUBSEP names[i]
    sort_array(ranked, "@val_str_desc")
    print "Top 5:"
    for (i = 1; i <= 5 && i <= n; i++) {
        split(ranked[i], p, SUBSEP)
        printf "  %s: %d\n", p[2], p[1] + 0
    }
}
```

6. Keyboard Shortcuts

Shortcut	Action
Ctrl+Enter	Execute the AWK script
Ctrl+F	Open find/replace in the script editor
Ctrl+Z	Undo in the script editor
Ctrl+Shift+Z	Redo in the script editor
Tab	Indent selected lines
Shift+Tab	Unindent selected lines
Ctrl+/	Toggle line comment

7. Browser Limitations

Because PenultiAWK runs in a browser rather than a shell environment, certain AWK features behave differently:

- **File I/O:** print > "file" and getline < "file" send output to and read from the output/input panels, not actual files.
- **Pipes:** print | "cmd" is not supported. Output goes to the output panel with a warning.
- **system():** Shell commands cannot be executed. system() prints a warning and returns -1.
- **ENVIRON:** The ENVIRON array is empty since browsers do not expose environment variables.
- **srand():** JavaScript does not support seeded random numbers, so srand() is a no-op. rand() uses Math.random().
- **Infinite loop protection:** Loops are limited to 1,000,000 iterations to prevent browser lockup.

8. Version History

Version	Changes
v0.04	Added asort(), asorti(), sort_array() built-in functions. Added Sort buttons on Input and Output panels.
v0.03	Added RFC 4180 CSV support (-csv flag, csv_split, csv_print, csv_printf). Added JSON support (-json flag, json_parse, json_stringify, JSONKEYS array). Auto-detection of JSON Lines and JSON array input formats.
v0.02	Broad POSIX AWK compatibility using a lexer/parser/AST/interpreter architecture. CodeMirror 6 editor. Extensive support for standard built-in functions, control flow, arrays, user-defined functions, printf, and getline, subject to browser limitations.
v0.01	Initial prototype.