

CSV Utils

Author: Norman Carver ©2022

Contents

1	Introduction	2
2	Utilities List	3
3	Usage Examples	5
4	The Utilities	8
4.1	csv-cat	8
4.2	csv-clean	10
4.3	csv-counts-fields	15
4.4	csv-counts-records	17
4.5	csv-cut	19
4.6	csv-edit	21
4.7	csv-format	23
4.8	csv-get-field	26
4.9	csv-get-record	28
4.10	csv-grep	30
4.11	csv-grep-cond	32
4.12	csv-grep-cond-records	34
4.13	csv-join	36
4.14	csv-names2nums	39
4.15	csv-paste	40
4.16	csv-print	43
4.17	csv-process-field	47
4.18	csv-process-records	52
4.19	csv-records	55
4.20	csv-reformat-date-time	58
4.21	csv-remove-duplicate-keys	61
4.22	csv-replace-field	63
4.23	csv-sort	65

4.24	csv-sort-mult	69
4.25	csv-validate	73
4.26	csv-to-dsv	75
4.27	dsv-to-csv	77
5	CSV File Syntax	79
6	General Behavior Notes	82
6.1	Header Records	82
6.2	File Arguments and Standard Input	82
6.3	Empty “CSV” Files	82
6.4	Blank (Empty) Records	83
7	Integrating CSV Utils with Standard Linux Utilities	84
8	AWK Code Basics	85
8.1	AWK Expressions vs Programs	85
8.2	Assignments to \$0	86
8.3	Formatting Real Numbers	86
8.4	Alternative Cases in AWK Code	86
8.5	AWK getline	87
9	Date-Time Format Strings	89
10	Build-Time Parameters	91
10.1	CSV File Size Limits	91
10.2	Embedded Characters Replacements	91
10.3	Utility Program Paths	92
11	Performance	93

1 Introduction

CSV Utils is a set of *command-line utilities* for manipulating *Comma Separated Values* (CSV) files. They are written in C and emphasize execution speed.

CSV files consist of a sequence of *records*, each record consisting of a sequence of *fields*. Records are separated by record separator/terminator character(s), fields are separated by single commas (, 's). The main standard for CSV files is *RFC 4180*, though it is possible to find “CSV files” that do not strictly adhere to RFC 4180.

The CSV Utils component programs are able to handle CSV files compatible with RFC 4180, but are often more flexible. For example, RFC 4180 specifies that all records are to have *identical numbers of fields*. However, it can sometimes be useful to produce “CSV files” having varying numbers of fields in records (avoiding the need to add empty padding fields). Because of this, most CSV Utils programs do not require that argument files have uniform field counts. If important, `csv-validate` can check for records with non-uniform field counts and `csv-clean` can both check for and *repair* such records.

In addition to the field structure of records, most CSV files are structured with the first record being a *header* record that names the fields. However, not every CSV file is guaranteed to have such a header (the first record may be a standard data record). Default operation of the CSV Utils programs will assume there is a header record that contains field names. However, a number of the CSV Utils programs have options that modify how first records in CSV files are treated, to be able to handle CSV files without header records.

When utilities require that field arguments be specified, all utilities accept *1-based field indices* by default. A number of the utilities will also accept field *names* (as found in a *header* record) with the `-Q` option. Our experience after a great deal work processing CSV files has been that field *names* can be problematic. Not only are field names likely to be much longer than indices, one must get the exact spelling, capitalization, and separators correct—often very annoying! It is easy to use “`csv-print 1 file.csv`” to get a quick mapping of header field names to indices, and not hard to remember that you want to operate on, e.g, field 5. By contrast, with field names we might have to remember whether the field name is: “`Last Name`”, “`Last_Name`”, “`last name`”, “`Name Last`”, etc, etc. And then there are the header typos (“`Last_Nam`”) and inconsistencies (e.g., “`Last Name`” but “`First_Name`”). Nonetheless, there are occasions when it is easier to specify a field name, instead of determining the index of the field in a CSV file with very many fields.

The utilities do not ever modify their argument CSV files. Instead, the results of operations are written to *standard output*. This means output must be *redirected* to be saved in a file (or directed through a *pipeline*). All error messages and informational messages are written to *standard error*.

The CSV Utils programs have generally been kept focused on a single operation, to be fast and easy to understand. Some desired operations on CSV files may thus require use of multiple programs. It is easy to combine multiple CSV Utils programs via shell *pipelines*, since all the programs can be used as Linux/UNIX “*filters*” (they *read from standard input* without file arguments and write to standard output). On modern *multicore* computers, pipeline stages run *in parallel*. This means that multiple pipelined programs will very likely run faster than a single, more complex utility program would.

2 Utilities List

The main set of CSV Utils programs includes:

- **csv-cat**: “cat” (print out and concatenate) one or more CSV files
- **csv-clean**: “clean” a CSV file (unquoting fields, removing embedded CR’s and LF’s, and/or fixing field counts in records)
- **csv-counts-fields**: print out field counts for the records
- **csv-counts-records**: print records counts for one or more CSV files
- **csv-cut**: “cut” (print out) specific fields from the records, to reduce dimensionality
- **csv-edit**: replace the value of one field in one record
- **csv-format**: produce readable, columnized text version of a CSV file
- **csv-grep**: “grep” a field, to select records with field matching a pattern
- **csv-grep-cond**: “grep” a field, to select records with field meeting a condition
- **csv-grep-cond-records**: “grep” a record, to select records with multiple fields meeting a joint condition
- **csv-join**: append records from a second CSV file onto the records from another CSV file, based on having matching values in fields of the records
- **csv-paste**: append records from a second CSV file onto the records from another CSV file, based on record order
- **csv-print**: print out the fields of one (or more) records, on separate lines for readability
- **csv-process-field**: process/modify a field of the records, using AWK code
- **csv-process-records**: process/modify entire records, using AWK code
- **csv-records**: print out a range of records
- **csv-reformat-date-time**: reformat date-time values in a field of the records
- **csv-remove-duplicate-keys**: remove records with duplicate key values
- **csv-replace-field**: apply search-replace patterns to a field of the records
- **csv-sort**: sort records based on the values in a single field
- **csv-sort-mult**: sort records based on the values in multiple fields
- **csv-validate**: validate a CSV file’s syntax and get stats on various properties

Two programs are intended for use in shell scripts:

- **csv-get-field**: print out a single field the from next record
- **csv-get-record**: print out the next record

One program converts a list of field *names* into a list of field numbers, so allows field names to be used when invoking the utilities:

- **csv-names2nums**: convert list of field names to field numbers

Two programs are for use in converting between CSV format and alternative DSV formats:

- **csv-to-dsv**: convert CSV file to DSV file with alternative delimiter char
- **dsv-to-csv**: convert DSV file with alternative delimiter char to CSV file

3 Usage Examples

This section provides a set of examples of common/useful usage patterns. See the sections for the relevant CSV Utils programs for further information and examples.

Remember that the utilities do not ever modify their argument CSV files. Instead, the results of operations are written to *standard output*. This means output must be *redirected* to be saved in a file.

- Get the *usage message* (use `--help` with any utility):
`csv-cat --help`
- Merge a set of monthly CSV files (months denoted 01, 02, ..., 12):
`csv-cat data-??.csv > data-all.csv`
- Remove unnecessary quoting from a CSV file:
`csv-clean -c filewquotes.csv > filewoquotes.csv`
- Identify records with incorrect numbers of fields (i.e., not same as header):
`csv-counts-fields -m file.csv`
- Show field count (only) of header:
`csv-counts-fields -cn1 file.csv`
- Determine number of records (including header) in all local CSV files:
`csv-counts-records *.csv`
- Reduce dimensionality of CSV file by removing unwanted fields:
`csv-cut 1-3,6,12-14 file.csv > file-reduced.csv`
- Move “key” field from field 5 to field 1:
`csv-cut 5,1-4,6- file.csv > file-reordered.csv`
- Get a field’s set of unique values (without header field name being included):
`csv-cut -o 3 file.csv | sort -u`
- Remove records where a particular field is empty:
(will also remove any “blank” records)
`csv-grep -Fvx 3 '' file.csv > file-woempties.csv`
- Remove records where *two* fields are empty:
`csv-grep -Fvx 3 '' file.csv | csv-grep -Fvx 9 '' > file-woempties.csv`
- Remove records where a particular field is empty or whitespace:
`csv-grep -vx 3 '[:space:]*' file.csv > file-woempties.csv`
- Get records (w/header) where field 2 has a value between 100 and 200:
(`csv-grep-cond` uses AWK code for conditions.)
`csv-grep-cond 2 '($0>=100)&&($0<=200)' file.csv`

- Get records where fields 5 and 10 have the same value:
(`csv-grep-cond-records` uses AWK code for conditions.)
`csv-grep-cond-records '$5==$10' file.csv`
- Join person records from two files using same keys:
`csv-join 1 people1.csv 1 people2.csv > combined.csv`
- Join records from two identically sorted files:
`csv-paste 1 people1.csv 1 people2.csv > combined.csv`
- Add a single field key that combines multiple fields:
(Combining last and first name fields as: Last_Name:First_Name.)
(Uses Bash *process substitution* to run `csv-cut` to get fields “on the fly.”)
`f=file.csv; csv-paste <(csv-cut -s: 2,3 "$f") "$f"`
- As above, but then remove fields that were combined:
`f=file.csv; csv-paste <(csv-cut -s: 2,3 "$f") "$f" | csv-cut 1,2,5-`
- View CSV file header, in easy to read format:
`csv-print 1 file.csv`
- View particular record along with field names, in easy to read format:
`csv-print -h 25 file.csv`
- Replace a field’s real number values with (rounded) integer values:
(`csv-process-field` uses AWK code for computation.)
`csv-process-field -e '{ $0="int($0+0.5)}' 3 file-wreals.csv > file-wints.csv`
- Change value of field 2 based on value of field 1:
(`csv-process-records` uses AWK code for computation.)
`csv-process-records -e '$1=="Jones"&&$2=="Anne"{$2="Annie"}' file.csv > file-new.csv`
- Create a CSV file with the first 100 records (plus header) of another:
`csv-records -hn100 2 file.csv > file100.csv`
- Get field 5’s value from record #200 of a CSV file:
`csv-records -n1 200 | csv-cut 5`
- Reformat date+time field 4 to ISO format:
`csv-reformat-date-time 4 '%D %T' '%F %T' dates.csv > dates-iso.csv`
- Remove patient records where updated records have been added:
(Patient IDs are field 1, file has been (stable) sorted on ID.)
`csv-remove-duplicate-keys 1 file-wdups.csv > file-wodups.csv`
- Add double quotes around a field’s values when unquoted:
(if CSV file is valid, unquoted fields will not contain embedded special characters)
`csv-replace-field -x 3 '[^"]*' '"&"' file.csv > file-wquotes.csv`
- Cut a field’s values to the first “subfield,” where separator is dot:
(E.g., cut codes like Q35.9 to “main” code Q35)
`csv-replace-field 3 -mx '([^.]+)(\..+)?' '&1' file.csv > file-mod.csv`

- Sort a file on a numeric key in field 2:
`csv-sort -n 2 file.csv > file-sortedbykey.csv`
- Sort a file on a date-time field with only a date:
(Requires supplying date-time format since no time in field.)
`csv-sort -D'%m/%d/%Y' 5 file.csv > file-sortedbydate.csv`
- Sort a file on numeric key field 2 first and date-time field 3 second:
`csv-sort-mult -k2n -k3d 5 file.csv > file-sortedbykeydate.csv`
- Check whether a CSV file contains embedded special characters:
`csv-validate unknown.csv`

4 The Utilities

4.1 csv-cat

Program to “cat” together one or more CSV files. Purpose is to merge a set of identically-structured CSV files into a single valid CSV file. Unlike standard `cat`, this program understands that CSV file *header records* need to be treated specially. By default, the header record from the *first* argument file only it output (so a valid CSV file is output when all CSV file arguments contain header records).

Usage: `csv-cat [OPTION...] [CSVFILE...]`

Options:

- `-b` -- blank records passed through (removed by default)
- `-h` -- header records should all be printed (not just from first CSVFILE)
- `-o` -- omit headers (don't print any header records)
- `-t` -- terminate output records with CR+LF (`\r\n`) (LF only by default)
- `-z` -- zero (null char) terminate output records (for use with `sort`, `uniq`, etc.)

Using this program to cat together multiple CSV files produces valid CSV output only if all CSV files share the same field structure (same number of fields, same meanings, same order). Note that no validation is performed to ensure all argument files have identical field counts. Because of this, the output may not represent a valid CSV format file if fed incompatible files.

There are additional uses for this program when applied to a single CSV file. For example, it can be used to strip headers off of a CSV file to start a pipeline (`-o` option).

By default, *blank records* are removed (not output). The `-b` option can be used to pass such records through. Note that if the initial record(s) are blank, and the `-b` option is not given, the header record will be considered the first non-blank record in each file.

The `-z` option allows `csv-cat` to be used to read/parse CSV files that contain *embedded linefeeds* (`\n`'s in quoted fields), and feed the resulting records to a program that cannot read such CSV files since the embedded linefeeds will be interpreted as record/line terminators. A number of GNU Linux/UNIX utilities have an option that will interpret “lines” as being terminated by *null chars* (`\0`'s) instead of linefeeds. See [Integration with Standard Utilities](#) for further information.

Examples of Using `csv-cat`:

- Merge a set of monthly CSV files (months denoted 01, 02, ..., 12), each having the same header and format:

```
csv-cat data-??.csv > data-all.csv
```

- Merge a set of monthly CSV files that do not have header records:

```
csv-cat -h data-??.csv > data-all.csv
```

- Start a pipeline without including CSV file header:

```
csv-cat -o file.csv | ...
```

- Use a pipeline with `tail` to print out the last five records in a CSV file: (Must use `tr` to fix record linefeeds.)

```
csv-cat -z test1.csv | tail -z -n5 | tr '\0' '\n'
```

4.2 csv-clean

Program to “clean” a CSV file in one or more of multiple possible ways:

1. “cleaning” fields (unquoting fields that do not need to be quoted);
2. padding/trimming records to make field counts consistent with header;
3. removing or padding blank records;
4. replacing embedded CR (`\r`) and LF (`\n`) characters with text;
5. quoting empty fields;
6. quoting or making empty, unquoted fields that are whitespace only
7. quoting or “cleaning,” unquoted fields that have leading/trailing whitespace

Usage: `csv-clean [OPTION...] [CSVFILE]`

Options:

```
-b -- blank records removed
-B -- blank records padded with empty fields (matching header)
-c -- clean quoted fields
    (remove double quotes and unescape embedded \"'s if possible)
-d -- delete/trim extra fields when records contain more fields than header
-e -- empty fields quoted (this includes padding fields)
-p -- pad records with empty fields when records contain less fields than header
-r -- replace embedded CR (\r) and LF (\n) characters with text
-sSEPARATOR -- separator string for output fields (comma by default),
    intended for use with -u option
-t -- terminate output records with CR+LF (\r\n) (LF only by default)
-u -- unquote quoted fields (remove surrounding \"'s and unescape embedded \"'s)
    Note: this can leave output as an invalid CSV, so best used with -s option
-v -- verbose: print final cleaning report
-V -- verbose: print warning/error messages about every CSV issue
-w -- whitespace fields quoted
-W -- whitespace fields made empty (can be combined with -e to quote)
-x -- leading/trailing whitespace fields quoted
-X -- leading/trailing whitespace in unquoted fields remove
```

Default behavior (no options) is to make no changes to the CSV file, but this isn’t useful, so select different types of cleaning operations with appropriate options. To guarantee a valid CSV file for output, use the `-b/-B`, `-d`, and `-p` options together. (Use [csv-validate](#) to assess validity and characteristics of a CSV file.)

Field “cleaning” behavior will convert a valid CSV file with unnecessarily quoted fields into a new valid CSV file with only those fields that must be quoted still quoted. This is useful for dealing with CSV files produced by systems that always quote all fields. A field in a CSV file *must* be quoted if the field contains any of the following characters:

- commas (,)
- double quotes (")
- carriage-returns (`\r`).
- linefeeds (`\n`)

Fields that contain these embedded characters are left unchanged by this program (unless the `-r` or `-u` options are given). “Cleaning” of quoted fields that do *not* contain these any of these embeds:

1. removes the surrounding double quotes;
2. converts embedded, *escaped/doubled* `"`'s (double-quotes) to unescaped/single `"`'s;

CSV files are to have identical numbers of fields in every record (see [CSV File Syntax](#)). This program can “fix” field counts so that every record contains the same number of fields: padding records deficient in fields with empty fields and trimming fields from records with extra fields.

Blank records are also not allowed in standard CSV files(see [CSV File Syntax](#)). This program can either remove blank records or pad them with empty fields, in order to produce a valid CSV file (see the `-b` and `-B` options).

Carriage-return and *linefeed* characters embedded in (quoted) fields can cause issues when using programs/commands that expect text files. The `-r` option can be used to convert such embedded CR/LF characters into text strings, allowing a CSV file to be processed by standard text file programs. Note that if embedded CR/LF characters are being replaced with text, this may allow more quoted fields to be unquoted by field “cleaning.”

The text strings that replace embedded CR/LF characters are set at compile time—see [Embedded Characters Replacements](#). It should be easy to substitute alternative strings using the `sed` utility in a pipeline (see examples below).

Empty fields (fields whose value is the “empty string”) can be confusing. The `-e` option makes empty values clear with double quotes. Note that the `-e` option interacts with the `-c` option with empty fields.

Fields that consist of *only whitespace* (spaces/tabs), are sometime produced when a value is lacking, but they are not the same as an *empty* field (“empty string”), so can cause issues for programs processing the CSV file data. The `-w` and `-W` options allow modifying such fields.

Unquoted fields that have *leading/trailing whitespace* characters (spaces/tabs) may be intended to have the whitespace ignored, but this is not in line with RFC 4180 nor how the CSV Utils programs behave. The `-x` and `-X` options allow making clear/correct how to deal with leading/trailing whitespace in fields.

The *unquote* (`-u`) option forces field unquoting, even if the result is an invalid CSV format field or a field that will make a record incorrect or ambiguous. It is best used only in conjunction with the `-s` option and a field separator that does not appear in any of the fields in the CSV file.

Detailed warning and error messages about CSV syntax can be shown with the `-V` option. Note that CSV syntax errors will cause termination (with failure exit status).

Examples of Using csv-clean:

- Turn invalid CSV file into one that meets RFC 4180:

```
csv-clean -bdp invalid.csv > valid.csv
```

- “Cleaning” a CSV file of unnecessary quoting:

```
csv-cat filewquotes.csv
```

```
"Field1","Field2","Field3","Field4","Field5"
"A 1 A","A 2 A","A 3 A","A 4 A","A 5 A"
"B 1 B","B 2 B","B 3 B","B 4 B","B 5 B"
"C 1 C","C 2 C","C 3 C","C 4 C","C 5 C"
```

```
csv-clean -c filewquotes.csv
```

```
Field1,Field2,Field3,Field4,Field5
A 1 A,A 2 A,A 3 A,A 4 A,A 5 A
B 1 B,B 2 B,B 3 B,B 4 B,B 5 B
C 1 C,C 2 C,C 3 C,C 4 C,C 5 C
```

- Remove blank records, i.e., to make CSV file valid:

```
csv-cat wblanks.csv
```

```
Field1,Field2,Field3,Field4,Field5
A 1 A,A 2 A,A 3 A,A 4 A,A 5 A

B 1 B,B 2 B,B 3 B,B 4 B,B 5 B
C 1 C,C 2 C,C 3 C,C 4 C,C 5 C
```

```
D 1 D,D 2 D,D 3 D,D 4 D,D 5 D
empty,,E 3 E,,E 5 E
spaces, ,F 3 F, ,F 5 F
tabs, ,G 3 G, ,G 5 G
both, ,H 3 H, ,H 5 H
```

```
csv-clean -b wblanks.csv
```

```
Field1,Field2,Field3,Field4,Field5
A 1 A,A 2 A,A 3 A,A 4 A,A 5 A
B 1 B,B 2 B,B 3 B,B 4 B,B 5 B
C 1 C,C 2 C,C 3 C,C 4 C,C 5 C
D 1 D,D 2 D,D 3 D,D 4 D,D 5 D
empty,,E 3 E,,E 5 E
spaces, ,F 3 F, ,F 5 F
tabs, ,G 3 G, ,G 5 G
both, ,H 3 H, ,H 5 H
```

- Pad blank records, i.e., to make CSV file valid:

```
csv-clean -B wblanks.csv

Field1,Field2,Field3,Field4,Field5
A 1 A,A 2 A,A 3 A,A 4 A,A 5 A
,,,,
B 1 B,B 2 B,B 3 B,B 4 B,B 5 B
C 1 C,C 2 C,C 3 C,C 4 C,C 5 C
,,,,
,,,,
D 1 D,D 2 D,D 3 D,D 4 D,D 5 D
empty,,E 3 E,,E 5 E
spaces, ,F 3 F, ,F 5 F
tabs, ,G 3 G, ,G 5 G
both, ,H 3 H, ,H 5 H
```

- Pad blank records and quote empty fields:

```
csv-clean -Be wblanks.csv

Field1,Field2,Field3,Field4,Field5
A 1 A,A 2 A,A 3 A,A 4 A,A 5 A
"","","","",""
B 1 B,B 2 B,B 3 B,B 4 B,B 5 B
C 1 C,C 2 C,C 3 C,C 4 C,C 5 C
"","","","",""
"","","","",""
D 1 D,D 2 D,D 3 D,D 4 D,D 5 D
empty,,E 3 E,,E 5 E
spaces, ,F 3 F, ,F 5 F
tabs, ,G 3 G, ,G 5 G
both, ,H 3 H, ,H 5 H
```

- Make CSV file with embedded CR/LF's compatible with standard Linux/UNIX text file utilities by replacing the embedded characters with text:

```
csv-clean -r file-wembeds.csv | wc -l
```

- Substitute alternative replacement strings using `sed`:

```
csv-clean -r file-wembeds.csv | sed -e 's/<CR>/return/g;s/<LF>/newline/g'
```

- Fixing file with nonuniform field counts, with messages and report:

```
csv-cat notuniform.csv
```

```
Field1,Field2,Field3,Field4,Field5
A 1 A,A 2 A,A 3 A,A 4 A,A 5 A
B 1 B,B 2 B,B 3 B,B 4 B,B 5 B,B 6 B
C 1 C,C 2 C,C 3 C,C 4 C
D 1 D,D 2 D,D 3 D,D 4 D,D 5 D,D 6 D
E 1 E,E 2 E,E 3 E,E 4 E,E 5 E
F 1 F,F 2 F,F 3 F
```

```
csv-clean -dpvV notuniform.csv
```

```
Field1,Field2,Field3,Field4,Field5
A 1 A,A 2 A,A 3 A,A 4 A,A 5 A
B 1 B,B 2 B,B 3 B,B 4 B,B 5 B
csv-clean: Warning: record #3 has more fields than header (6 vs 5), trimming
C 1 C,C 2 C,C 3 C,C 4 C,
csv-clean: Warning: record #4 has less fields than header (4 vs 5), padding
D 1 D,D 2 D,D 3 D,D 4 D,D 5 D
csv-clean: Warning: record #5 has more fields than header (6 vs 5), trimming
E 1 E,E 2 E,E 3 E,E 4 E,E 5 E
F 1 F,F 2 F,F 3 F,,
csv-clean: Warning: record #7 has less fields than header (3 vs 5), padding
```

Cleaning Report:

Number of records read in total:	7
Number of header fields:	5
Number of fields cleaned:	0
Number of fields unquoted:	0
Number of records with fields padded:	2
Number of records with fields trimmed:	2
Number of blank records removed:	0
Number of blank records padded:	0
Number of empty fields quoted:	0
Number of embedded CRs replaced:	0
Number of embedded LFs replaced:	0

4.3 csv-counts-fields

Program to print out and check fields counts for records in a CSV file. When checking field counts, the fields count for the header record, record #1, is taken as the correct count.

Usage: `csv-counts-fields [OPTION...] [CSVFILE]`

Options:

```
-c          -- counts only, do not print record number prefixes
-fFIRST    -- first record to print fields count for (1-based index, header is 1, the default)
-h         -- print header (record #1) fields count even if header not in records range
-m         -- mismatches only: print fields counts for records that differ from header
-nNUMBER   -- number of records to check (default is all from FIRST)
-v         -- verbose: mark records where fields count differs from header
```

By default, shows counts for header (record 1) and all records, prefixed by record number. Options allow instead showing:

- counts only for records where the counts are incorrect (differ from header);
- counts for a limited range of records;
- bare counts (without record number prefixes);

Examples of Using csv-counts-fields:

- Show field counts of all records, with record number prefixes:

```
csv-counts-fields notuniform.csv
```

```
1:5
```

```
2:5
```

```
3:6
```

```
4:4
```

```
5:6
```

```
6:5
```

```
7:3
```

- Show field counts of all records, marking incorrect counts:

```
csv-counts-fields -v notuniform.csv
```

```
1:5
```

```
2:5
```

```
3:6*
```

```
4:4*
```

```
5:6*
```

```
6:5
```

```
7:3*
```

- Show counts only for records with incorrect counts:

```
csv-counts-fields -m notuniform.csv
```

```
3:6
```

```
4:4
```

```
5:6
```

```
7:3
```

- Show field count (only) of header:

```
csv-counts-fields -c -n1 file.csv
```

```
5
```

- Show field count (only) of record #4:

```
csv-counts-fields -c -f4 -n1 notuniform.csv
```

```
4
```

- Show field counts of select records, and include header:

```
csv-counts-fields -h -f4 -n3 notuniform.csv
```

```
1:5
```

```
4:4
```

```
5:6
```

```
6:5
```

- Show field counts of select records, marking incorrect counts:

```
csv-counts-fields -v -f4 -n3 notuniform.csv
```

```
4:4*
```

```
5:6*
```

```
6:5
```

4.4 csv-counts-records

Program to print out counts of the number of records in one or more CSV files.

Usage: csv-counts-records [OPTION...] [CSVFILE...]

Options:

- h -- header records should NOT be counted (counted by default)
- v -- not verbose: do NOT print filenames to prefix counts

Examples of Using csv-counts-records:

- Show count of records in a file, counting header:

```
csv-counts-records test1.csv
6:test1.csv
```

- Show count of records in a file, not counting header:

```
csv-counts-records -h test1.csv
5:test1.csv
```

- Show count of records in a file, count only:

```
csv-counts-records -v test1.csv!
6
```

- Show counts of records in multiple files:

```
csv-counts-records test?.csv
6:test1.csv
99:test2.csv
24:test3.csv
```

4.5 csv-cut

Program to “*cut*” (print out) only specified fields from the records in a CSV file. Purpose is to reduce the dimensionality of a CSV file, by producing a CSV file with only the needed subset of fields. Can also be used to reorder fields, e.g., so a “key” field occurs first.

Usage: `csv-cut [OPTION...] FIELDS_LIST [CSVFILE]`

FIELDS_LIST must be a comma-separated list of 1-based field index specs: numbers (e.g., 3), ranges (e.g., 3-6), or unbounded range (e.g., 3-) as final entry, or header field names if -Q option.

Options:

- a -- allow FIELDS_LIST specs to refer to fields beyond those in the header (may create empty fields)
- b -- blank records passed through (removed by default, see also -1)
- e -- error terminate if FIELDS_LIST entries missing from record (replaced with empty fields by default)
- l -- limit fields for unbounded FIELDS_LIST spec (e.g., 5-) to number of header fields
- o -- omit header field(s) from output
- Q -- query header to get indices of FIELDS_LIST field names
- r -- reverse meaning of FIELDS_LIST: print out all fields EXCEPT those specified
- sSEPARATOR -- output field separator (comma by default)
- t -- terminate output records with CR+LF (\r\n) (LF only by default)
- w -- warn about missing fields and fields not in header (with -a option)
- z -- zero (null char) terminate output records (for use with sort, uniq, etc.)
- 1 -- interpret blank record as having a single empty field (see also -b)

Similar in purpose to the Linux/UNIX `cut` utility, but this program works properly with all CSV files, since it handles *quoted* CSV fields that can contain embedded commas (,), carriage-returns (\r's), and/or linefeeds (\n's).

Since CSV Utils generally aim to produce valid CSV output, `csv-cut`'s default behavior is as follows:

- the number of fields in the header is taken as the standard for the CSV output;
- FIELDS_LIST specs outside of the number of header fields will cause an error (see -a option to allow);
- FIELDS_LIST fields missing from a record will result in *empty fields* in the output records;
- *blank records* are *removed* (see -b option to pass through);
- -1 option causes blank records to be interpreted as having a single *empty field* (such records would be ambiguous in “CSV files” with single fields);

An unbounded FIELDS_LIST spec like “4-” will result in empty fields being added to records with less than the number of header fields, and fields beyond the number of header fields being output in records with such fields (see -l option).

Examples of Using `csv-cut`:

- Remove unwanted fields (reduce dimensionality) of a CSV file:
`csv-cut 1,3,6-8,10,18- alldata.csv > keydata.csv`
- Reorder the fields of a CSV file:
`csv-cut 5,12-14,11,1-4,6-10,15- alldata.csv > reordered.csv`
- Get a field's set of unique values (without header name being included):
`csv-cut -o 5 | sort -u`
- Sum the values in an integer field of a CSV file:
(Skips header, uses AWK to do the computation.)
`csv-cut -o 8 data.csv | awk -e '/[0-9]+/{sum+=$0};END{print sum}'`
- Extract a single field out of a CSV file, using the field name:
`csv-cut -Q ID data.csv > IDs.csv`
- Create a data file from a numeric field in a CSV file:
`csv-cut -oQ Age data.csv > ages.dat`
- Create a new file excluding single private field, using the field name:
`csv-cut -rQ SSN personnel.csv > personnel-anonymized.csv`

4.6 csv-edit

Program to edit (replace) the value of *one field* in *one record* in a CSV file. I.e., this program allows you to edit a single field in a CSV file, without having to open the file in a spreadsheet program or text editor.

Usage: `csv-edit [OPTION...] RECORD FIELD NEWVALUE [CSVFILE]`

RECORD is the 1-based record number to operate on.

FIELD is the 1-based field index to operate on.

NEWVALUE is the replacement value of the specified field.

(NEWVALUE can contain '&' to be substituted by the previous value of the field.)

Options:

`-t --` terminate output records with CR+LF (`\r\n`) (LF only by default)

The CSV Utils are not intended to be used for minor editing of CSV files, such as correcting a field value in one or two records, etc. Nonetheless, `csv-edit` was added to make it easy to replace the value of *one field in one record*. More involved editing of individual fields is best accomplished using *spreadsheet programs* like LibreOffice Calc, or even a *text editor*.

Examples of Using `csv-edit`:

- Change header field #3:

```
csv-edit 1 3 CSVFILE > CSVFILE-NEW
```

- Adding double quotes around field 3 in record 15:

```
csv-edit 15 3 '"&"' CSVFILE > CSVFILE-NEW
```

- Including a literal ampersand in a new field value:

```
csv-edit 1 5 'Street_Number\&Name' CSVFILE > CSVFILE-NEW
```

4.7 csv-format

Program to print out a CSV file formatted in a readable manner, as text. That is, to produce *columnized* text from the contents of the fields.

Usage: `csv-format [OPTION...] [CSVFILE]`

Options:

```
-a          -- allow CSVFILE to have nonuniform field counts
-b          -- blank records passed through (removed by default)
-c          -- center fields (left justify by default)
-fWIDTH    -- fixed field width to use for all fields
-mWIDTH    -- minimum field width to use, overrides automatically determined widths
-nNUMRECS  -- number of records to use to determine field widths (10 by default)
-r          -- right justify fields (left by default)
-sSEPARATOR -- separator between field values (single space by default)
-u          -- unquote quoted fields before printing: remove surrounding "'s,
              unescape embedded "'s, and replace embedded CR's/LF's with text
-wWIDTHS   -- comma-separated list of fields widths
              e.g., -w10,12,10,15,20 (must have enough values for all records)
-xWIDTH    -- field width to use for extra fields (beyond header or -w spec)
```

The default separator between fields is just a single space, to keep lines as short as possible. Readability can be improved by using the `-s` option to have longer or more visible separator strings.

If output is too long to view on the “terminal” (output lines get wrapped), redirect the output to a file and use a text editor to view it.

By default, maximum field widths will be determined by examining the first 10 records (including the header). More or less records can be examined using the `-n` option. Alternatively, field widths can be manually specified with the `-f` or `-w` options.

Records are required to have uniform numbers of fields (i.e., same as header) unless the `-a` option is given. If the `-a` option needs to be used due to records with additional fields, one will have to make sure a width to use for the extra fields is known. This might be done via the `-f`, `-w`, or `-x` options.

Since quoting is done simply to make CSV files with embedded commas/carriage-returns/linefeeds unambiguous, quoted fields do not represent the “true contents” of a field, so use of the unquote (`-u`) option may be desirable. This option will strip the double quotes surrounding each field, unescape embedded double-quotes, and replace any embedded CR's/LF's with text so readable.

Sample CSV Files:

- `test.csv`:

```
Field1,"Field2",Field3,"Field4",Field5
A 1 A,"A2",A 3 A 3 A,"A 4 A,""4"",A",5 A
B 1 B,"B2",B 3 B 3 B,"B 4 B,""4"",B",5 B
C 1 C,"C2",C 3 C 3 C,"C 4 C,""4"",C",5 C
```

- Note that:

- Field #2 is quoted, but contains no embeds.
- Field #4 is quoted, and non-header records contain embedded commas and (escaped) double-quotes

Examples of Using `csv-format`:

- Basic formatting, auto width determination:

```
csv-format test.csv
```

```
Field1 "Field2" Field3      "Field4"          Field5
A 1 A  "A2"      A 3 A 3 A  "A 4 A,""4"",A" 5 A
B 1 B  "B2"      B 3 B 3 B  "B 4 B,""4"",B" 5 B
C 1 C  "C2"      C 3 C 3 C  "C 4 C,""4"",C" 5 C
```

- Increasing spacing between fields:

```
csv-format -s'      ' test.csv
```

```
Field1      "Field2"      Field3      "Field4"          Field5
A 1 A      "A2"          A 3 A 3 A  "A 4 A,""4"",A" 5 A
B 1 B      "B2"          B 3 B 3 B  "B 4 B,""4"",B" 5 B
C 1 C      "C2"          C 3 C 3 C  "C 4 C,""4"",C" 5 C
```

- Unquoting fields for readability:

```
csv-format -us'      ' test.csv
```

```
Field1      Field2      Field3      Field4          Field5
A 1 A      A2          A 3 A 3 A  A 4 A,"4",A 5 A
B 1 B      B2          B 3 B 3 B  B 4 B,"4",B 5 B
C 1 C      C2          C 3 C 3 C  C 4 C,"4",C 5 C
```

- Centering output within the columns:

```
csv-format -cus'      ' test.csv
```

```
Field1      Field2      Field3      Field4          Field5
A 1 A      A2          A 3 A 3 A  A 4 A,"4",A 5 A
B 1 B      B2          B 3 B 3 B  B 4 B,"4",B 5 B
C 1 C      C2          C 3 C 3 C  C 4 C,"4",C 5 C
```

- Right justifying output within the columns:

```
csv-format -rus'      ' test.csv
```

Field1	Field2	Field3	Field4	Field5
A 1 A	A2	A 3 A 3 A	A 4 A,"4",A	5 A
B 1 B	B2	B 3 B 3 B	B 4 B,"4",B	5 B
C 1 C	C2	C 3 C 3 C	C 4 C,"4",C	5 C

- Using fixed field width:

```
csv-format -f15 test.csv
```

Field1	"Field2"	Field3	"Field4"	Field5
A 1 A	"A2"	A 3 A 3 A	"A 4 A,""4"",A"	5 A
B 1 B	"B2"	B 3 B 3 B	"B 4 B,""4"",B"	5 B
C 1 C	"C2"	C 3 C 3 C	"C 4 C,""4"",C"	5 C

- Specifying field widths:

```
csv-format -uw10,10,20,20,10 test.csv
```

Field1	Field2	Field3	Field4	Field5
A 1 A	A2	A 3 A 3 A	A 4 A,"4",A	5 A
B 1 B	B2	B 3 B 3 B	B 4 B,"4",B	5 B
C 1 C	C2	C 3 C 3 C	C 4 C,"4",C	5 C

- Using alternative field separator string (unquoted fields):

```
csv-format -us:: test.csv
```

Field1::Field2::Field3	::Field4	::Field5
A 1 A ::A2	::A 3 A 3 A::A 4 A,"4",A::	5 A
B 1 B ::B2	::B 3 B 3 B::B 4 B,"4",B::	5 B
C 1 C ::C2	::C 3 C 3 C::C 4 C,"4",C::	5 C

- Using tab separators via Bash “C strings” (works well given consistent field widths):

```
csv-format -us$'\t' test.csv
```

Field1	Field2	Field3	Field4	Field5
A 1 A	A2	A 3 A 3 A	A 4 A,"4",A	5 A
B 1 B	B2	B 3 B 3 B	B 4 B,"4",B	5 B
C 1 C	C2	C 3 C 3 C	C 4 C,"4",C	5 C

4.8 csv-get-field

Program to “get” (print to stdout) the value of a single field from the next CSV record from *standard input*. It is intended to be called in scripts and pipelines, replacing the use of AWK or similar to extract CSV fields. Reads *next record* from standard input and writes the specified field to *standard output* (followed by a linefeed).

Usage: `csv-get-field [OPTION...] FIELD`

FIELD is the 1-based index of the field to get/print.

Options:

- c -- clean quoted field if result is still a valid CSV field
(remove double quotes and unescape embedded "'s)
- e -- error terminate if FIELD does not exist in a record
- u -- unquote quoted FIELD (remove surrounding "'s and unescape embedded "'s);
-u takes precedence over -c if both supplied;
Note: can return an invalid CSV field
- v -- verbose: print any CSV syntax error messages (suppressed by default)
- w -- print warning if FIELD does not exist in a record

By default, this program prints an empty string value (i.e., outputs blank line) if the specified field does not exist in the next record. This behavior can be modified using `-e` and `-v` options (see above).

Note that each run of this utility reads just the *next* CSV record from standard input and outputs the single field value from that record. The *offset* for standard input will be advanced to the end of the next CSV record only. Because of this, `csv-get-field` can be used in a loop to move through a CSV file (see below examples).

`csv-cut` can perform the same basic field extraction action as `csv-get-field`, but has more complicated syntax and will be slightly slower to extract a single field from a record. Since `csv-cut` automatically loops through all records in the argument file/stream, it is directly usable as a Linux/UNIX “filter” program, to pass on only a particular field(s) from a stream of CSV records. Using `csv-get-field` as a pipeline filter would require embedding it in shell loop. The intended use for `csv-get-field` is to exact a single field from a record that has already been read or otherwise obtained (see below examples).

A *failure* exit status is returned if `stdin` is at “file-end”, a CSV parsing error occurs, or the specified field is missing and the `-e` option was given.

Example Shell Script Usage of csv-get-field:

Use in shell script to obtain CSV record fields, when CSV records read/obtained separately:

```
#!/usr/bin/bash
... get CSV record into variable record ...
firstname=$(echo "$record" | csv-get-field 1)
lastname=$(echo "$record" | csv-get-field 2)
id=$(echo "$record" | csv-get-field 5)
gpa=$(echo "$record" | csv-get-field 12)
... use extracted record information ...
```

Use in a shell script that loops through CSV file records using Bash `read`:

```
#!/usr/bin/bash
datefield=$1
csvfile=$2
...
recnum=1
while read record; do
    # Skip header:
    if [[ $recnum == 1 ]]; then continue; fi

    date=$(echo "$record" | csv-get-field $datefield)
    ...use variable date as desired...

    recnum=$((recnum+1))
done < "$csvfile"
...
```

Note that above script will have problems with CSV files that contain embedded linefeeds (`\n`'s)! See example below, or use [csv-get-record](#) instead.

Alternative shell script that shows how to use `csv-get-field` to loop through CSV file records getting single fields out of each:

```
#!/usr/bin/bash
datefield=$1
csvfile=$2
...
recnum=1
while date=$(csv-get-field "$datefield"); do
    # Skip header:
    if [[ $recnum == 1 ]]; then continue; fi

    ...use variable date as desired...

    recnum=$((recnum+1))
done < "$csvfile"
...
```

4.9 csv-get-record

Program to “get” (print to stdout) the next CSV record from a CSV file stream passed to it via *standard input*. It is intended to be called in scripts and pipelines, replacing the use of AWK or similar to extract CSV fields. Reads *next record* from standard input and writes the entire record to *standard output*. Each record that is output is followed by termination string: linefeed (`\n`) by default, CR+LF (`\r\n`) with `-t` option, or null char (`\0`) if `-z` option.

Usage: `csv-get-record [OPTION...]`

Options:

- `-t` -- terminate output records with CR+LF (`\r\n`) (LF only by default)
- `-v` -- verbose: print any CSV syntax error messages (suppressed by default)
- `-z` -- zero (null char) terminate output records (for use with `sort`, `uniq`, etc.)

Reads just the next CSV records from standard input, outputting that single record for each call. The *offset* for standard input will be advanced to the end of the next CSV record only. Because of this, `csv-get-record` can be used in a loop to move through a CSV file (see below example).

A *failure* exit status is returned if `stdin` is at “file-end” or a CSV parsing error occurs. This makes the program compatable with use in a shell `while` loop, for example.

Example Shell Script Usage of csv-get-record:

Read through all records in a CSV file:

```
#!/usr/bin/bash
csvfile=$1
...
while record=$(csv-get-record); do
    ... use variable record as desired ...
done < "$csvfile"
```

4.10 csv-grep

Program to “*grep*” a particular *field* in a CSV file, to select/remove records where field matches a pattern. Can handle *regex* and *fixed string* patterns (fixed string only when patterns come from a file). Options are similar to standard `grep`.

Usage: `csv-grep [OPTION...] FIELD PATTERN [CSVFILE]`

Usage: `csv-grep [OPTION...] -fPATTERN_FILE FIELD [CSVFILE]`

FIELD is the 1-based index of the field to grep, or header field name if -Q option. PATTERN is the regex or string pattern to match.

(PATTERN assumes extended regex syntax, so do not escape regex metachars).

Options:

- b -- blank records passed through (ignored by default, see also -1)
- fFILE -- read fixed string PATTERNS from FILE
- F -- interpret PATTERN as a fixed string
- h -- process header record (passed through by default)
- i -- ignore case when matching PATTERN
- mNUM -- stop after NUM matches
- n -- prefix each line of output with the 1-based line number
- o -- omit header record (not grep'd, not passed through)
- q -- suppress all output, look at exit status
- Q -- query header to get index of FIELD name
- t -- terminate output records with CR+LF (\r\n) (LF only by default)
- u -- unquote quoted FIELD before applying grep pattern
(remove surrounding '’s and unescape embedded \”’s)
- v -- invert match (i.e., select non-matching)
- w -- warn about records lacking FIELD (such records are ignored)
- x -- exactly match the (entire) field with PATTERN
- 1 -- interpret blank record as having a single empty field (see also -b)

Uses Linux/UNIX regex utilities, with POSIX *Extended Regular Expression* option. This means that there is no need to escape regex metacharacters in regex patterns. See “*man 7 regex*” for more info.

By default, *header* records are *passed through* automatically to the output. The -h and -o options can be used to change this behavior.

Records that lack FIELD are *ignored* (not tested or passed through), since they cannot be grep'd. The -w option will cause warnings about such records.

Blank records are also *ignored* (not tested or passed through), since they won't have FIELD, so cannot be grep'd. See, however, the -1 option.

This program does *not* use sophisticated matching algorithms as (GNU) `grep` does. This is reasonable since with most CSV files, field values to match against tend to be relatively short, so highly efficient matching algorithms that require preprocessing of patterns are unlikely to result in much/any speedup.

Examples of Using `csv-grep`:

- Remove records where field 2 is *empty*:
(will also remove any “blank” records)
`csv-grep -Fvx 2 '' file.csv > file-woempty.csv`
- Remove records where field 2 is “*blank*” (empty/whitespace):
`csv-grep -vx 2 '[:space:]*' file.csv > file-woblanks.csv`
- As previous, but using field name:
`csv-grep -vxQ ID '[:space:]*' file.csv > file-woblank-ids.csv`
- Get records (plus header) where year in date field 5 is ≥ 2000 :
`csv-grep 5 '20[0-9]{2}' file.csv > file-2000.csv`
- Get records (w/header) where field 3 is a (positive) integer:
`csv-grep -x 3 '[0-9]+' file.csv > file-wints.csv`
- Get records (w/header) where field 3 is not a negative number:
`csv-grep -v '^-[0-9]+' file.csv > file-wonegs.csv`
- Produce a data file with all positive integer values from field 3:
`csv-grep -ox 3 '[0-9]+' file.csv | csv-cut 3 > field.data`
- Get records (w/header) where field 8 contains the word “diagnosis” (any case):
`csv-grep -Fi 8 diagnosis medical.csv > diagnosis.csv`
- Get records (w/header) where IDs in field 1 match those in file `ids.txt`:
(Use `-u` to deal with field values possibly being quoted.)
`csv-grep -ux -fids.txt 1 medical.csv > patients.csv`
- Get records (w/header) where IDs in field 1 do *not* match any of those in file `ids.txt`:
`csv-grep -vx -fids.txt 1 medical.csv > patients.csv`

Multiple fields may be selected on by *pipelining* multiple `csv-grep` calls together:

```
csv-grep -x 1 '[0-9]+' CSVFILE | csv-grep -vx 2 '' > CSVFILE-SELECT
```

(Produce a new CSV file where one field 1 contains only integers and field 2 is not empty.)

Entire records (i.e., multiple fields) can be selected on using standard `grep`—as long as the CSV file does not contain embedded linefeeds (`\n`'s): `grep PATTERN CSVFILE > CSVFILE-SELECT`

If the CSV file contains *embedded linefeeds*, a pipeline can be used with the `-z` option of `csv-records`:
`csv-records -z 1 CSVFILE-WEMBEDS | grep -z PATTERN | tr '\0' '\n' > CSVFILE-SELECT`

4.11 csv-grep-cond

Program to “*grep*” a particular *field* in a CSV file, to select/remove records where field meets some *condition*. Options are consistent with standard **grep**. A *condition* is specified using an AWK **Expression**: AWK code that evaluates to a value. See the AWK info below, plus the [AWK Code](#) section, for more information on AWK.

Usage: `csv-grep-cond [OPTIONS] FIELD CONDITION [CSVFILE]`

FIELD is the 1-based index of the field to grep, or header field name if -Q option.

Options:

- b -- blank records passed through (ignored by default, see also -1)
- h -- process header record (passed through by default)
- lLIB_FILE -- load AWK functions/library file LIB_FILE
- mNUM -- stop after NUM matches
- n -- prefix each line of output with the 1-based line number
- o -- omit header record (not grep'd, not passed through)
- q -- quiet: suppress all output, look at exit status
- Q -- query header to get index of FIELD name
- t -- terminate output records with CR+LF (\r\n) (LF only by default)
- u -- unquote quoted FIELD before testing CONDITION
(remove surrounding "’s and unescape embedded "’s)
- v -- invert match (i.e., select non-matching)
- w -- warn about records lacking FIELD (such records are ignored)
- 1 -- interpret blank record as having a single empty field (see also -b)

CONDITION is to be a single AWK Expression, i.e., AWK code that results in a value. The AWK code will receive FIELD’s values in successive CSVFILE records as input “records.” E.g., CSV record “A,2,C” and FIELD of 2, will result in CONDITION having \$0 be “2”. Records where CONDITION’s value is 1 (or other non-zero) are considered matched. Records where CONDITION’s value is 0 (or “”) are considered non-matched. Example CONDITION: ‘(\$0>=10 && \$0<=20) || (\$0>100)’

Since general AWK expression code—including standard and user-defined functions—can be used to represent a condition, this program can implement more sophisticated evaluation of a field than [csv-grep](#). However, **csv-grep** is simpler and faster when regex or substring patterns suffice.

By default, *header* records are *passed through* automatically to the output. The -h and -o options can be used to change this behavior.

Records that lack FIELD are *ignored* (not tested or passed through), since they cannot be grep’d. The -w option will cause warnings about such records.

Blank records are also *ignored* (not tested or passed through), since they won’t have FIELD, so cannot be grep’d. See, however, the -1 option.

Examples of Using `csv-grep-cond`:

- Get records (w/header) where field 2 is between 100 and 200:
`csv-grep-cond 2 '($0>=100)&&($0<=200)' file.csv`
- Using user-defined library functions:
`csv-grep-cond -lmathfuncs.awk 2 'abs($0)<=10' file.csv`
- Using AWK string functions:
(Select records where field 5 has more than 10 characters or digits.)
`csv-grep-cond 5 'length($0)>10' file.csv`
- Selecting on *multiple fields* by *pipelining* multiple `csv-grep-cond` calls together:
(Produces a new CSV file where field 1 is greater than 100 and field 2 is less than 200.)
`csv-grep-cond 1 '$0>100' file.csv | csv-grep-cond 2 '$0<200' > select.csv`

AWK Coding Requirements for `csv-grep-cond`:

- Code must be a valid AWK *condition*.
- Does not modify `RS` or `ORS`.
- Does not call `print` or `printf`.
- Does not call `next` or `nextfile`.

AWK Expressions:

The `FIELD` for each successive CSV record will be available in the AWK `CONDITION` code as `$0`. `CONDITION` must be an AWK *expression*: AWK code that evaluates to a *value*. The resulting value of the expression code for each record determines whether the record is considered matched or not matched:

- values of 1 (or any other *non-zero* value) are considered *matched*;
- values of 0 (or `""`) are considered *not matched*;

Note that AWK *expressions* are not AWK “*programs*”. AWK program code cannot be used as `CONDITION`, an AWK error will result. See the [AWK Code](#) section for more information on AWK.

4.12 csv-grep-cond-records

Program to “*grep*” records in a CSV file, to select/remove records that meet some *condition*. Options are consistent with standard **grep**. A *condition* is specified using an AWK **Expression**: AWK code that evaluates to a value. See the AWK info below, plus the [AWK Code](#) section, for more information on AWK.

Usage: `csv-grep-cond-records [OPTIONS] CONDITION [CSVFILE]`

Options:

- `-b -- blank records grep'd (removed by default)`
- `-h -- process header record (passed through by default)`
- `-lLIB_FILE -- load AWK functions/library file LIB_FILE`
- `-mNUM -- stop after NUM matches`
- `-n -- prefix each line of output with the 1-based line number`
- `-o -- omit header record (not grep'd, not passed through)`
- `-q -- quiet: suppress all output, look at exit status`
- `-t -- terminate output records with CR+LF (\r\n) (LF only by default)`
- `-v -- invert match (i.e., select non-matching)`

CONDITION is to be a single AWK Expression, i.e., AWK code that results in a value. The AWK code will receive each successive CSVFILE record as an input “record” (\$0). I.e., each CSVFILE record will be available as \$0, with its fields as \$1, \$2, etc. Records where CONDITION’s value is 1 (or other non-zero) are considered matched. Records where CONDITION’s value is 0 (or “”) are considered non-matched.
Example CONDITION: `'$1>=20 || ($2!="0" && $3!="Admit")'`

Unlike [csv-grep-cond](#), conditions here can evaluate *multiple fields* of a record. While *multiple fields* may be *and*-selected on by *pipelining* [csv-grep-cond](#) calls together, this program can implement more complex logic involving multiple fields of records. Nonetheless, [csv-grep-cond](#) will be faster when evaluating a single field. Likewise, [csv-grep](#) will be the fastest when regex or substring patterns (applied to a single field) suffice for selection.

By default, *header* records are *passed through* automatically to the output. The `-h` and `-o` options can be used to change this behavior.

Blank records in CSVFILE are *ignored* (not tested or passed through). See, however, the `-b` option.

Examples of Using `csv-grep-cond-records`:

- Get records (w/header) where field 2 is greater than 100 and field 3 is less than 200:
`csv-grep-cond-records '($2>100)&&($3<200)' file.csv`
- Using user-defined library functions:
`csv-grep-cond-records -lmathfuncs.awk '(abs($1)<=10)|| (int($5)==$5)' file.csv`
- Get records where fields 5 and 10 have the same value:
`csv-grep-cond-records '$5==$10' file.csv`

AWK Coding Requirements for `csv-grep-cond-records`:

- Code must be a valid AWK *condition*.
- Does not modify `RS`, `FS`, `ORS`, `OFS`, `FPAT`.
- Does not call `print` or `printf`.
- Does not call `next` or `nextfile`.

AWK Expressions:

The fields of each successive CSV record will be available in the AWK `CONDITION` code as `$1`, `$2`, etc. `CONDITION` must be an AWK *expression*: AWK code that evaluates to a *value*. The resulting value of the expression code for each record determines whether the record is considered matched or not matched:

- values of 1 (or any other *non-zero* value) are considered *matched*;
- values of 0 (or `""`) are considered *not matched*;

Note that AWK *expressions* are not AWK “*programs*”. AWK program code cannot be used as `CONDITION`, an AWK error will result. See the [AWK Code](#) section for more information on AWK.

4.13 csv-join

Program to “*join*” (append) records from a second CSV file with “*matching*” records in a first CSV file. Records are considered to *match* if their key field values are *identical*. Output is a valid CSV file with fields from both argument CSV files.

Usage: `csv-join [OPTION...] KEYFIELD1 CSVFILE1 KEYFIELD2 [CSVFILE2]`

KEYFIELD1 is the 1-based index of the key field in CSVFILE1,
or header field name if `-Q` option.

KEYFIELD2 is the 1-based index of the key field in CSVFILE2,
or header field name if `-Q` option.

Options:

- `-a` -- allow CSVFILE2 to lack matching records
- `-o` -- identically ordered keys in files, so can be efficient finding match in CSVFILE2
when using `-a` option (Note: all CSVFILE2 keys must match a CSVFILE1 key!)
- `-Q` -- query header to get index of FIELD name
- `-sSEPARATOR` -- separator between CSVFILE records (comma by default)
- `-t` -- terminate output records with CR+LF (`\r\n`) (LF only by default)
- `-u` -- unquote quoted KEYFIELDS before comparing
(remove surrounding "’s and unescape embedded "’s)
- `-v` -- verbose: print final report about join operations

To have a predictable output, *key values must be unique in each file*. I.e., there must be at most a single record with each key value in both files. If keys are not unique in either file, the results will depend on the particular order records occur in the files. In other words, the resulting CSV is not predictable, and likely to not make sense.

CSVFILE1 is considered the base CSV file to which information is being added. The output is to be a valid CSV file, with the same number of records as CSVFILE1, and all records must have the exact same number of fields: the sum of the number of fields in CSVFILE1 plus those in CSVFILE2 (headers taken as the standard counts). Note that this implies that the key fields from both files get included in the output. This is done because the fields may have different names or may be desired as reference points in the final output. Should only one instance be wanted, this can easily be achieved by using `csv-cut` (e.g., in a pipeline after `csv-join`).

By default, it is required that there be a matching record in CSVFILE2 for each record in CSVFILE1. There can, however, be “extra” records in CSVFILE2 (i.e., whose keys do not match records in CSVFILE1). Such records are ignored (since, again, CSVFILE1 is considered the base CSV file being added to).

If some records in CSVFILE1 will not have matches in CSVFILE2, the `-a` option must be used. In this case, unmatched records in CSVFILE1 will have an appropriate number of empty fields added to ensure the resulting output is valid CSV format (all records with same number of fields).

The CSV files do *not* need to be *sorted* on the key fields to use this function. However, speed will be fastest if the files are sorted identically on the key fields. With CSV files with many records, the speed difference can be very significant. This means that it may be faster to use `csv-sort` to create argument files before running this program. One can also use Bash *process substitution* to avoid creating intermediate sorted files (see examples below).

When some records in `CSVFILE1` may not have matches in `CSVFILE2` and the `-a` option is used, having the keys in the files ordered identically can greatly increase efficiency, since it will avoid having to search `CSVFILE2` for every non-existent match. When the files have their keys ordered identically, use the `-o` option with the `-a` option. Be aware, however, that with these two options in use, *all CSVFILE2 keys must match a CSVFILE1 key!*

Sample CSV files:

- `people1.csv`:

```
ID,Last_Name,First_Name,Street,Apt,City,State,Zip
111111111,Smith,John,111 Main St,,Chicago,IL,66601
444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304
333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209
222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001
```

- `people2.csv` (note extra ID record):

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City
444444444,Harris,Beth,F,10/30/1987,St. Louis
333333333,Carter,Robert,M,05/23/2000,Denver
555555555,Smith,Susan,F,06/3/1955,New York
111111111,Smith,John,M,03/15/1945,Chicago
222222222,Jones,Mary,F,12/01/1990,Memphis
```

Examples of Using `csv-join`:

- Basic join of two files:

```
csv-join 1 people1.csv 1 people2.csv
```

```
ID,Last_Name,First_Name,Street,Apt,City,State,Zip,ID,Last_Name,First_Name,Sex,Birthdate,Birth City
111111111,Smith,John,111 Main St,,Chicago,IL,66601,111111111,Smith,John,M,03/15/1945,Chicago
444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304,444444444,Harris,Beth,F,10/30/1987,St. Louis
333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209,333333333,Carter,Robert,M,05/23/2000,Denver
222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001,222222222,Jones,Mary,F,12/01/1990,Memphis
```

- Join two files as above, but then sort on key and remove duplicated fields:

```
csv-join 1 people1.csv 1 people2.csv | csv-sort -hn 1 | csv-cut 1-8,12-
```

```
ID,Last_Name,First_Name,Street,Apt,City,State,Zip,Sex,Birthdate,Birth City
111111111,Smith,John,111 Main St,,Chicago,IL,66601,M,03/15/1945,Chicago
222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001,F,12/01/1990,Memphis
333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209,M,05/23/2000,Denver
444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304,F,10/30/1987,St. Louis
```

- Join of two files, first file has record with unmatched key:

```
csv-join 1 people2.csv 1 people1.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,ID,Last_Name,First_Name,Street,Apt,City,State,Zip
444444444,Harris,Beth,F,10/30/1987,St. Louis,444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304
333333333,Carter,Robert,M,05/23/2000,Denver,333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209
555555555,Smith,Susan,F,06/3/1955,New York
Error: CSVFILE2 does not have matching record for key: 555555555
```

- Same as last example, except allow unmatched records with `-a` option:

```
csv-join -a 1 people2.csv 1 people1.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,ID,Last_Name,First_Name,Street,Apt,City,State,Zip
444444444,Harris,Beth,F,10/30/1987,St. Louis,444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304
333333333,Carter,Robert,M,05/23/2000,Denver,333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209
555555555,Smith,Susan,F,06/3/1955,New York,,,,,
111111111,Smith,John,M,03/15/1945,Chicago,111111111,Smith,John,111 Main St,,Chicago,IL,66601
222222222,Jones,Mary,F,12/01/1990,Memphis,222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001
```

- Same as last example, except **incorrectly** adding `-o` option even though files not identically ordered:

```
csv-join -ao 1 people2.csv 1 people1.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,ID,Last_Name,First_Name,Street,Apt,City,State,Zip
444444444,Harris,Beth,F,10/30/1987,St. Louis,,,,,
333333333,Carter,Robert,M,05/23/2000,Denver,,,,,
555555555,Smith,Susan,F,06/3/1955,New York,,,,,
111111111,Smith,John,M,03/15/1945,Chicago,111111111,Smith,John,111 Main St,,Chicago,IL,66601
222222222,Jones,Mary,F,12/01/1990,Memphis,222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001
```

- Same as last example, except using Bash *process substitution* to ensure identically ordered records:

```
csv-join -ao 1 <(csv-sort 1 people2.csv) 1 <(csv-sort 1 people1.csv)
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,ID,Last_Name,First_Name,Street,Apt,City,State,Zip
111111111,Smith,John,M,03/15/1945,Chicago,111111111,Smith,John,111 Main St,,Chicago,IL,66601
222222222,Jones,Mary,F,12/01/1990,Memphis,222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001
333333333,Carter,Robert,M,05/23/2000,Denver,333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209
444444444,Harris,Beth,F,10/30/1987,St. Louis,444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304
555555555,Smith,Susan,F,06/3/1955,New York,,,,,
```

4.14 csv-names2nums

Program to convert a list of CSV file field *names* to the field *numbers* (as used by other CSV Utils programs).

Usage: `csv-names2nums [OPTIONS...] FIELD_NAMES_LIST CSVFILE`

OPTIONS:

`-n -- output linefeed (\n) after output list (just list by default)`

The input and output lists are to be *comma-separated* lists, e.g.: `Last_Name,First_Name,SS_NUM,5,6,37`. They can also be a single field name or number.

Note that this program reads the header from its `CSVFILE` argument. Because of this, it *cannot be used as part of a pipeline* and requires a CSV file argument (not standard input).

Examples of Using `csv-names2nums`:

- Determining single field's number:

```
csv-names2nums SS_Num file.csv
37
```

- Determining list of field numbers:

```
csv-names2nums Last_Name,First_Name,SS_Num file.csv
5,6,37
```

- Using field names with `csv-cut` instead of using the `-Q` option:

```
csv-cut $(csv-names2nums Last_Name,First_Name,SS_Num file.csv) file.csv
```

- Storing a field number in a Bash variable (so can reuse repeatedly):

```
field=$(csv-names2nums SS_Num file.csv)
csv-grep $field 123456789 file.csv
...
```


4.15 csv-paste

Program to “*paste*” (i.e., append) corresponding records from a second CSV file onto the records from a first CSV file. It is intended to be used to combine data for the same set of keys/individuals, from two separate CSV files, where *the records in both files are sorted identically on the keys/individuals*.

Usage: `csv-paste [OPTION...] CSVFILE1 [CSVFILE2]`

Options:

- a -- allow CSVFILE2 records to have variable numbers of fields
- m -- mismatches of record counts allowed, CSVFILE1 > CSVFILE2 (must be equal by default)
- M -- mismatches of record counts allowed, CSVFILE1 < CSVFILE2 (must be equal by default)
- p -- pad CSVFILE2 records with empty fields when records contain less fields than header
- sSEPARATOR -- separator between CSVFILE records (comma by default)
- t -- terminate output records with CR+LF (\r\n) (LF only by default)
- v -- verbose: print final report about paste operations
- 1 -- interpret blank record as having a single empty field

CSVFILE1 is considered the base CSV file to which information is being added. By default, CSVFILE1 and CSVFILE2 are to have identical numbers of records, and all records in CSVFILE2 are to have identical numbers of fields. Running this command would then result in a valid CSV format output, with the same number of records as both of the input CSV files, where each record has the same number of fields (the sum of the number of fields in CSVFILE1 plus those in CSVFILE2).

With the -m option, CSVFILE2 is allowed to have *less* records than CSVFILE1. Unmatched CSVFILE1 records will be output followed by SEPARATOR only. If the -p option is supplied, empty fields for the missing CSVFILE2 records will be added.

With the -M option, CSVFILE2 is allowed to have *more* records than CSVFILE1. The “extra” records in CSVFILE2 will simply be ignored.

Both CSV file arguments should be valid CSV files, meaning all records in each file should have an identical number of fields. However, the -a option allows pasting records with *variable numbers of fields* from CSVFILE2 onto the records of CSVFILE1, since this is occasionally useful with some types of data. In such a case, the resulting output will not be a valid standard CSV file due to the resulting records having variable numbers of fields.

The -v option will cause a report on `csv-paste` operations to be output (to *standard error*) at the end.

Sample CSV Files:

- people1.csv:

```
ID,Last_Name,First_Name,Street,Apt,City,State,Zip
111111111,Smith,John,111 Main St,,Chicago,IL,66601
222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001
333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209
444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304
```

- people2.csv:

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City
111111111,Smith,John,M,03/15/1945,Chicago
222222222,Jones,Mary,F,12/01/1990,Memphis
333333333,Carter,Robert,M,05/23/2000,Denver
444444444,Harris,Beth,F,10/30/1987,St. Louis
```

- people2b.csv:

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City
111111111,Smith,John,M,03/15/1945,Chicago
222222222,Jones,Mary,F,12/01/1990,Memphis
333333333,Carter,Robert,M,05/23/2000,Denver
444444444,Harris,Beth,F,10/30/1987,St. Louis
555555555,Biden,Joe,M,01/15/1910,Scranton
666666666,Jones,Jack,M,04/23/2001,Chicago
```

Examples of Using csv-paste:

- Basic paste of two identically sorted files:

```
csv-paste people1.csv people2.csv
```

```
ID,Last_Name,First_Name,Street,Apt,City,State,Zip,ID,Last_Name,First_Name,Sex,Birthdate,Birth City
111111111,Smith,John,111 Main St,,Chicago,IL,66601,111111111,Smith,John,M,03/15/1945,Chicago
222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001,222222222,Jones,Mary,F,12/01/1990,Memphis
333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209,333333333,Carter,Robert,M,05/23/2000,Denver
444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304,444444444,Harris,Beth,F,10/30/1987,St. Louis
```

- As above, but then remove duplicate fields:

```
csv-paste people1.csv people2.csv |csv-cut 1-8,12-
```

```
ID,Last_Name,First_Name,Street,Apt,City,State,Zip,Sex,Birthdate,Birth City
111111111,Smith,John,111 Main St,,Chicago,IL,66601,M,03/15/1945,Chicago
222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001,F,12/01/1990,Memphis
333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209,M,05/23/2000,Denver
444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304,F,10/30/1987,St. Louis
```

- Add a single field key that includes multiple fields, to a CSV file:
(uses Bash *process substitution* with csv-cut)

```
csv-paste <(csv-cut -s: 2,3 people1.csv) people1.csv
```

```
Last_Name:First_Name,ID,Last_Name,First_Name,Street,Apt,City,State,Zip
Smith:John,111111111,Smith,John,111 Main St,,Chicago,IL,66601
Jones:Mary,222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001
Carter:Robert,333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209
Harris:Beth,444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304
```

- Paste of two identically sorted files, where first has extra records (error!):

```
csv-paste people2b.csv people1.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth_City,ID,Last_Name,First_Name,Street,Apt,City,State,Zip
111111111,Smith,John,M,03/15/1945,Chicago,111111111,Smith,John,111 Main St,,Chicago,IL,66601
222222222,Jones,Mary,F,12/01/1990,Memphis,222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001
333333333,Carter,Robert,M,05/23/2000,Denver,333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209
444444444,Harris,Beth,F,10/30/1987,St. Louis,444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304
555555555,Biden,Joe,M,01/15/1910,Scranton,
csv-paste: ERROR: mismatch @record #6: CSVFILE1 has more records than CSVFILE2
```

- Using -m option to eliminate above error and pad missing records:

```
csv-paste -mp people2b.csv people1.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth_City,ID,Last_Name,First_Name,Street,Apt,City,State,Zip
111111111,Smith,John,M,03/15/1945,Chicago,111111111,Smith,John,111 Main St,,Chicago,IL,66601
222222222,Jones,Mary,F,12/01/1990,Memphis,222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001
333333333,Carter,Robert,M,05/23/2000,Denver,333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209
444444444,Harris,Beth,F,10/30/1987,St. Louis,444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304
555555555,Biden,Joe,M,01/15/1910,Scranton,,,,,
666666666,Jones,Jack,M,04/23/2001,Chicago,,,,,,
```

- Paste of two identically sorted files, where second has extra records (error!):

```
csv-paste people1.csv people2b.csv
```

```
ID,Last_Name,First_Name,Street,Apt,City,State,Zip,ID,Last_Name,First_Name,Sex,Birthdate,Birth_City
111111111,Smith,John,111 Main St,,Chicago,IL,66601,111111111,Smith,John,M,03/15/1945,Chicago
222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001,222222222,Jones,Mary,F,12/01/1990,Memphis
333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209,333333333,Carter,Robert,M,05/23/2000,Denver
444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304,444444444,Harris,Beth,F,10/30/1987,St. Louis
./csv-paste: ERROR: mismatch @record #6: CSVFILE1 has less records than CSVFILE2
```

- Using -M option to eliminate above error:

```
csv-paste -M people1.csv people2b.csv
```

```
ID,Last_Name,First_Name,Street,Apt,City,State,Zip,ID,Last_Name,First_Name,Sex,Birthdate,Birth_City
111111111,Smith,John,111 Main St,,Chicago,IL,66601,111111111,Smith,John,M,03/15/1945,Chicago
222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001,222222222,Jones,Mary,F,12/01/1990,Memphis
333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209,333333333,Carter,Robert,M,05/23/2000,Denver
444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304,444444444,Harris,Beth,F,10/30/1987,St. Louis
```

4.16 csv-print

Program to print out a range of records in a CSV file, *printing each field on a separate line*, making it easy to read the values for each field.

Usage: `csv-print [OPTION...] FIRST_RECORD [CSVFILE]`

FIRST_RECORD is 1-based index of first record to print, header being record 1.
FIRST_RECORD may be "L" to print the last record in each file..

Options:

`-h` -- include header field name from CSVFILE when printing fields (when FIRST_RECORD>1)
`-nNUM` -- number of records to print (1 by default)

By default, only a single record is printed, but multiple records can be printed with the `-n` option. Then, each record's number is indicated, and records are separated by a blank line.

Field values are surrounded by *single quotes* (') to make the entire value clear, even when the field is empty or contains whitespace, carriage-returns, or linefeeds. (To print select CSV records as they appear in a file, use [csv-records](#).)

Fields are numbered, but field names (from the header record) can also be prepended by using the `-h` option.

Examples of Using csv-print:

- View header:

```
csv-print 1 people1.csv
1: 'ID'
2: 'Last_Name'
3: 'First_Name'
4: 'Street'
5: 'Apt'
6: 'City'
7: 'State'
8: 'Zip'
```

- View a particular record:

```
csv-print 4 people1.csv
1: '333333333'
2: 'Carter'
3: 'Robert'
4: '333 Mountain Rd'
5: ''
6: 'Denver'
7: 'CO'
8: '78209'
```

- View a particular record with header field names:

```
csv-print -h 4 people1.csv
1: ID: '333333333'
2: Last_Name: 'Carter'
3: First_Name: 'Robert'
4: Street: '333 Mountain Rd'
5: Apt: ''
6: City: 'Denver'
7: State: 'CO'
8: Zip: '78209'
```

- View last record in a file:

```
csv-print -h L ../csv-test-files/people1.csv
Record #5:
1: ID: '444444444'
2: Last_Name: 'Harris'
3: First_Name: 'Beth'
4: Street: '4444 County Rd 34'
5: Apt: ''
6: City: 'Springfield'
7: State: 'MO'
8: Zip: '52304'
```

- View multiple records:

```
csv-print -hn2 4 people1.csv
Record #4:
1: ID:      '333333333'
2: Last_Name: 'Carter'
3: First_Name: 'Robert'
4: Street:   '333 Mountain Rd'
5: Apt:      ''
6: City:     'Denver'
7: State:    'CO'
8: Zip:      '78209'

Record #5:
1: ID:      '444444444'
2: Last_Name: 'Harris'
3: First_Name: 'Beth'
4: Street:   '4444 County Rd 34'
5: Apt:      ''
6: City:     'Springfield'
7: State:    'MO'
8: Zip:      '52304'
```

- View headers in multiple files:

```
csv-print 1 people?.csv
==> people1.csv <==
1: 'ID'
2: 'Last_Name'
3: 'First_Name'
4: 'Street'
5: 'Apt'
6: 'City'
7: 'State'
8: 'Zip'

==> people2.csv <==
1: 'ID'
2: 'Last_Name'
3: 'First_Name'
4: 'Sex'
5: 'Birthdate'
6: 'Birth City'
```

- Viewing records in file with non-uniform numbers of fields:

```
csv-print -hn3 2 nonuniform.csv
```

```
Record #2:                <<correct number of fields (matches header)
```

```
1: Field1: 'A 1 A'
```

```
2: Field2: 'A 2 A'
```

```
3: Field3: 'A 3 A'
```

```
4: Field4: 'A 4 A'
```

```
5: Field5: 'A 5 A'
```

```
Record #3:                <<has one more field than header
```

```
1: Field1: 'B 1 B'
```

```
2: Field2: 'B 2 B'
```

```
3: Field3: 'B 3 B'
```

```
4: Field4: 'B 4 B'
```

```
5: Field5: 'B 5 B'
```

```
6: [EXTRA]: 'B 6 B'
```

```
Record #4:                <<has one less field than header
```

```
1: Field1: 'C 1 C'
```

```
2: Field2: 'C 2 C'
```

```
3: Field3: 'C 3 C'
```

```
4: Field4: 'C 4 C'
```

```
5: Field5: [MISSING]
```

4.17 csv-process-field

Program to *process/manipulate* values in a single field of the records in a CSV file, using provided *AWK program code*. The AWK program can modify the value of the field in select records, producing modified records that will be output in place of the originals. AWK program code can be supplied as a command-line argument if short and not reused frequently, or it can be supplied in AWK *source code file(s)* if longer or to be reused.

Usage: `csv-process-field [OPTIONS] FIELD [CSVFILE]`

`FIELD` is the 1-based field index to process, or header field name if `-Q` option.

AWK program code must be supplied via the `-e` or `-f` option.

Options:

<code>-b</code>	-- blank records passed through (removed by default, see also <code>-1</code>)
<code>-eCODE</code>	-- <code>CODE</code> is the AWK program to execute
<code>-fCODE_FILE</code>	-- <code>CODE_FILE</code> is a file containing the AWK program to execute
<code>-h</code>	-- process header record (passed through by default)
<code>-lLIB_FILE</code>	-- load AWK library/functions file <code>LIB_FILE</code>
<code>-o</code>	-- omit header record (not processed, not passed through)
<code>-Q</code>	-- query header to get index of <code>FIELD</code> name
<code>-t</code>	-- terminate output records with CR+LF (<code>\r\n</code>) (LF only by default)
<code>-1</code>	-- interpret blank record as having a single empty field (see also <code>-b</code>)

The AWK code will receive `FIELD`'s value in each `CSVFILE` record as its input "record" (`$0`). E.g., `CSVFILE` record "A,B,C" with `FIELD` of 2, will result in AWK code having `$0` be "B". Modify `FIELD`'s value by having the code do an assignment to `$0`: `$0=newvalue`.

Note that `$0` must have a string value, so use `sprintf()` to assign new numeric values. `FIELD` values in output records will be the values of `$0` produced for each `CSVFILE` record. AWK code is not to do any output (do not call `print` or `printf`). AWK code must be valid AWK "program," i.e., one or more rules: `[pattern]{actions}`.

The input CSV file is required to have `FIELD` in every record, and will terminate with an error if it is missing. This means that *blank records* are *not* allowed in CSV files input to this program. (Remove with [csv-clean](#) before using this program.)

Note that *the AWK code must be a valid AWK program*. I.e., it is to consist of one or more AWK *rules*, of the form: `[pattern]{actions}`. Invalid AWK programs will result in AWK errors or unwanted behavior.

Each CSV record's `FIELD` value is input to the AWK code as a *single AWK input "record"* (`$0`). E.g., with `FIELD` of 2 and CSV record "A,B,C", the AWK code will have `$0` be "B". Changes to `FIELD` are to be made with assignments to `$0`: `{...$0=newvalue}`. The value of `FIELD` in each output CSV record will be the value of `$0` upon completion of the AWK code processing of each AWK input record (i.e., `FIELD` in each original `CSVFILE` record).

AWK code should modify only `$0`, and it should do this only in case `FIELD`'s value is to be changed in a CSV record. See the below *AWK Coding Requirements* and *Modifying FIELD with AWK* subsections for more details about the requirements and functioning of AWK code for use with

this program. Read the **AWK Code** section of this manual for more information on AWK. Pay particular attention to the issues discussed in the subsection **Assignments to \$0**.

Utilizes system-installed AWK, which is assumed to be GNU AWK, run in a subprocess. Invalid AWK code passed to `csv-process-field` will cause error messages from the AWK subprocess, and termination. Fix the coding errors and rerun.

Examples of Using csv-process-field:

- Process field with mix of (positive) integers and reals, converting reals to integers by rounding:

```
csv-process-field -e '/^[0-9]+\.[0-9]+$/{'$0=sprintf("%d",int($0+0.5))}'  
3 data.csv > data-rounded.csv
```

- Convert 2-digit years to 4-digit years, taking into account correct century:
(See [Date-Time Format Strings](#) for further information.)

```
csv-process-field -f fixyear.awk 2 2digityrs.csv > 4digityrs.csv
```

fixyear.awk:

```
# Dates of form: month/dayofmonth/year[ time]  
/^[0-9]+\/[0-9]+\/[0-9]{2}((^[0-9].*)|())$/ {  
    pos = index($0, " ")  
    if (pos != 0)  
        date = substr($0,1,pos-1)  
    else  
        date = $0  
    split(date,parts,"/")  
    yr = parts[3]  
    if (yr >= 30)  
        #1930 to 1999  
        yr += 1900  
    else  
        #2000 to 2029  
        yr += 2000  
    if (pos != 0)  
        $0 = sprintf("%s/%s/%4d%s",parts[1],parts[2],yr,substr($0,pos))  
    else  
        $0 = sprintf("%s/%s/%4d",parts[1],parts[2],yr)  
}
```

- Unquote quoted fields that can be unquoted:
(I.e., unquote fields without embedded commas or dquotes.)

```
csv-process-field -f unquote.awk 2 file.csv > file-unquoted.csv
```

unquote.awk:

```
/^[^,]*$/ {  
    noquotes = substr($0,2,length($0)-2)  
    gsub(/"/,"\\\"",noquotes)  
    $0 = noquotes  
}
```

- Unquote using a function from an AWK library file (file with function definitions):

```
csv-process-field -l csv-functions.awk -e '{$0=unquote_field($0)}'
2 file.csv > file-unquoted.csv
```

- Change age (integer years) into age category (string):

```
csv-process-field -f agecat.awk 5 people.csv > people.agecat.csv
```

agecat.awk:

```
/^[0-9]+$/{
    if ($0 <= 18) $0 = "0-18"
    else if ($0 <= 39) $0 = "19-39"
    else if ($0 <= 59) $0 = "40-59"
    else $0 = "60+"
}

# Handle blank fields:
/^ *$/{
    $0 = "unknown"
}
```

AWK Coding Requirements for `csv-process-field`:

- Code must be a valid AWK *program*.
- Does not modify `RS` or `ORS`.
- Does not call `print` or `printf`.
- Does not call `next` or `nextfile`.

Modifying `FIELD` with AWK code:

The new value for `FIELD` in each CSV record will be the value of `$0` after the AWK code runs on the CSV record. `FIELD`'s value will be unchanged in a CSV record if `$0` is unchanged by the AWK code. The main way to change the value of `FIELD` in CSV records is by having the AWK code assign a new value to `$0`: e.g. `$0=newvalue`. One tricky aspect of this is that assignments to `$0` *must be done with string values!* Make sure to read the AWK subsection [Assignments to `\$0`](#) for more information on this issue.

`csv-process-field` vs. `csv-process-records`:

`csv-process-field` is designed to evaluate and modify only a *single* field in CSV records. By contrast, [csv-process-records](#) is designed to evaluate and modify *multiple* fields in CSV records. Here are some example tasks that `csv-process-records` can do that `csv-process-field` cannot:

- Modify one field's value based on the value of one or more other field(s).
- Create a new field that is a combination of multiple fields.

`csv-process-field` vs. `csv-replace-field`:

The use of AWK code with `csv-process-field` allows numeric, string, and other general types of manipulation of field values. By contrast, [csv-replace-field](#) allows only simple textual manipulation (search for pattern and replace). Here are some examples of differences in functionality between the two utilities:

- Field values that may be quoted: `csv-process-field` could properly unquote (including unescaping embedded "'s), while `csv-replace-field` could only remove surrounding double quotes, but not unescape embedded dquotes.
- Field values that are ages: `csv-process-field` could do something like converting an age into an age category (e.g., $56 \rightarrow 40-59$), `csv-replace-field` could not do this.
- Field values that are real numbers: `csv-process-field` could *round* the reals to integers, `csv-replace-field` could only *drop* the fractional component.

4.18 csv-process-records

Program to *process/manipulate* the records of a CSV file, using provided *AWK code*. The AWK code can modify fields in records, producing modified records that will be output in place of the originals.

Usage: `csv-process-records [OPTIONS] [CSVFILE]`

AWK program code must be supplied via the `-e` or `-f` option.

Options:

- `-aNEWFIELDS` -- append NEWFIELDS list to output header (use when adding fields)
- `-b` -- blank records processed (removed by default)
- `-eCODE` -- CODE is the AWK program to execute
- `-fCODE_FILE` -- CODE_FILE is a file containing the AWK program to execute
- `-h` -- process header record (passed through by default)
- `-lLIB_FILE` -- load AWK library/functions file LIB_FILE
- `-o` -- omit header record (not processed, not passed through)
- `-t` -- terminate output records with CR+LF (`\r\n`) (LF only by default)

The AWK code will receive each successive CSVFILE record as an input "record" (`$0`). I.e., each CSVFILE record will be available as `$0`, with its fields as `$1`, `$2`, etc. Modify a record's field by having the code do an assignment to the field: `$2=newvalue`. Output records will be the values of `$0` produced for each CSVFILE record. AWK code is not to do any output (do not call `print` or `printf`). AWK code must be valid AWK "program," i.e., one or more rules: `[pattern]{actions}`.

AWK code can be supplied as a command-line argument if short and not reused frequently, or it can be supplied in AWK *source code file(s)* if longer or to be reused.

Note that *the AWK code must be a valid AWK program*. I.e., it is to consist of one or more AWK *rules*, of the form: `[pattern]{actions}`. Invalid AWK programs will result in AWK errors or unwanted behavior.

Each CSV record is input to the AWK code as a *single AWK input "record:"* `$0` will be the entire CSV record, `$1`, `$2`, etc., will be the CSV record's fields. Changes are to be made through *assignments* to AWK field components: e.g., `{...$2=newvalue}`. The output CSV records will be the values of `$0` upon completion of the AWK code processing of each AWK input record (i.e., each original CSVFILE record).

See the below *AWK Coding Requirements* and *Modifying Records with AWK* subsections for more details about the requirements and functioning of AWK code for use with this program.

Utilizes system-installed AWK, which is assumed to be GNU AWK, run in a subprocess. Invalid AWK code passed to `csv-process-records` will cause error messages from the AWK subprocess, and termination. Fix the coding errors and rerun.

Examples of Using `csv-process-records`:

- Change value of one field based on value in another field:

```
csv-process-records -e '$1=="Jones"&&$2=="Anne"{$2="Annie"}'
file.csv > file-new.csv
```

- Exchange fields 1 and 2 (just an example, easier to use `csv-cut`):

```
csv-process-records -e '{temp=$1;$1=$2;$3=temp}' file.csv > file-new.csv
```

- Add a new field that is the sum of fields 1 and 3:

```
csv-process-records -a "Sum" -e '{$(NF+1)=$1+$3}' file.csv > file-new.csv
```

(Note use of `-a` option and `NF` variable to number new field.)

- Previous example without using `-a` option, using code in file `faddnr.awk`:

```
csv-process-records -h -f faddnr.awk file.csv > file-plus.csv
```

`faddnr.awk`:

```
NR==1 {$(NF+1) = "Sum"}
```

```
NR>1 {$(NF+1) = $1 + $3}
```

(Note use of `-h` option, and `NR` variable to select header vs. non-header processing.)

AWK Coding Requirements for `csv-process-record`:

- Code must be a valid AWK *program*.
- Does not modify `RS`, `FS`, `ORS`, `OFS`, `FPAT`.
- Does not call `print` or `printf`.
- Does not call `next` or `nextfile`.

Modifying Records with AWK code:

Each CSV record will be available in the AWK code as `$0`, with fields available as `$1`, `$2`, etc. Processing code can modify field values by assigning new values to them: e.g., `{...$2=newvalue}`. The possibly modified record from AWK processing will become the updated CSV record.

If CSV output is desired, `$0` should generally not be modified directly (e.g, don't do: `{...$0=...}`). However, this program can be used to do things like producing “data” files from the records in a CSV file. For example, if one wants a data file containing the averages of numeric fields 2, 3, and 5, this could be accomplished with AWK code like: `{ $0=sprintf("%.2f", ($2+$3+$5)/3.0) }`.

Make sure to read the AWK subsection [Assignments to \\$0](#) if writing code where `$0` is to be modified!

New fields can even be added by assigning to fields greater than `NF`: e.g., `{...$(NF+1)=newfieldvalue}`. Of course, if field(s) are to be added to records, new header field(s) should also be added. This will require using the `-h` option (see below) and an AWK rule for handling just the header (pattern: `NR==1`) and one or more other rules that will not affect the header (pattern: `NR>1`).

If the `-h` option is used, the header will be passed to AWK. In this case, the header will have `NR` of 1, while the first non-header record will have `NR` of 2. Without the `-h` option, the header record from the CSV file is passed through by default, meaning it is not counted by AWK, so `NR` will be 1 (one) for the first non-header record.

If `NF` has its value made smaller (`NF=Nf-2`), this will cause field(s) to be removed from the (ends of the) CSV records. Of course, if field(s) are to be removed from records, header field(s) should also be adjusted (so use the `-h` option and multiple rules, as discussed above).

See the [AWK Code](#) section for more information about AWK coding.

`csv-process-records` vs. `csv-process-field`:

`csv-process-records` is designed to evaluate and modify *multiple* fields in CSV records. By contrast, `csv-process-field` is designed to evaluate and modify only a *single* field in CSV records. Here are some example tasks that `csv-process-records` can do that `csv-process-field` cannot:

- Modify one field's value based on the value of one or more other field(s).
- Create a new field that is a combination of multiple fields.

4.19 csv-records

Program to print out a range of records in a list of CSV files. Records are output exactly as in argument CSV file, with the exception of possibly changing the record terminator characters (and removing blank records).

Usage: `csv-records [OPTION...] FIRST_RECORD [CSVFILE...]`

`FIRST_RECORD` is 1-based index of first record to print, header being record 1.
`FIRST_RECORD` may be "L" to print the last record in the file.

Options:

- `-b` -- blank records passed through (removed by default)
- `-h` -- print header even if `FIRST_RECORD` is not 1 (i.e., add header to output)
- `-nNUM` -- number of records to print (rest of file by default)
(does not count header if `-h` option)
- `-t` -- terminate output records with CR+LF (`\r\n`) (LF only by default)
- `-v` -- verbose: prefix records with linenums, and filenames if multiple CSVFILES
- `-z` -- zero (null char) terminate output records (for use with sort, uniq, etc.)

This program is primarily intended to extract ranges of records in CSV files, to e.g., start a processing pipeline working on a subsequence of CSV records. By default, the program prints the remaining records in each file argument starting from specified first record. The `-n` option allows limiting the number of records printed out (starting with the first record).

It accepts multiple CSV file arguments to allow for comparison of particular record(s) from multiple files, with the `-v` option useful for visual comparisons. If one desires to print out all the records from *multiple* CSV files, concat'ing the files into a valid CSV file, [csv-cat](#) should be used instead.

By default, header record(s) are not printed if `FIRST_RECORD` is not 1, but this behavior can be changed with the `-h` option.

By default, *blank records* are removed (and not counted against `NUM`). The `-b` option can be used to allow blank records to be passed through if desired (i.e., treated as valid records). Note that if the "header" record is blank, the `-h` option will use it as the header even without the `-b` option. Similarly, if `FIRST_RECORD` is L, to print the last record in the file, the last record will be printed even if it is blank.

The `-z` option allows `csv-records` to be used to read/parse CSV files that contain *embedded linefeeds* (`\n`'s in quoted fields), and feed the resulting records to a program that cannot read such CSV files (because the embedded linefeeds will be interpreted as record/line terminators). A number of GNU Linux/UNIX utilities have an option that will interpret "lines" as being terminated by *null chars* (`\0`'s) instead of linefeeds. See [Integration with Standard Utilities](#) for further information.

Examples of Using csv-records:

- Print out entire file (with LF only terminators):

```
csv-records 1 people1.csv
```

```
ID,Last_Name,First_Name,Street,Apt,City,State,Zip
111111111,Smith,John,111 Main St,,Chicago,IL,66601
222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001
333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209
444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304
```

- Print out entire file *without header*:

```
csv-records 2 people1.csv
```

```
111111111,Smith,John,111 Main St,,Chicago,IL,66601
222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001
333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209
444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304
```

- Print file starting at record 4 (non-header record #3):

```
csv-records 4 people1.csv
```

```
333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209
444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304
```

- Print file starting at record 4, with header:

```
csv-records -h 4 people1.csv
```

```
ID,Last_Name,First_Name,Street,Apt,City,State,Zip
333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209
444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304
```

- Print just record 4 (non-header record #3):

```
csv-records -n1 4 people1.csv
```

```
333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209
```

- Print just record 4, with header:

```
csv-records -hn1 4 people1.csv
```

```
ID,Last_Name,First_Name,Street,Apt,City,State,Zip
333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209
```

- Print last record in file, with record number prefix:

```
csv-records -v L people1.csv
```

```
5:444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304
```

- Print header records from multiple files to compare:

```
csv-records -vn1 1 people?.csv
```

```
people1.csv:1:ID,Last_Name,First_Name,Street,Apt,City,State,Zip
```

```
people2.csv:1:ID,Last_Name,First_Name,Sex,Birthdate,Birth City
```

```
people3.csv:1:ID,Last_Name,First_Name,Sex,Birthdate,Birth City,Weight_Lbs
```

- Print records from multiple files to compare:

```
csv-records -vn2 2 people1*.csv
```

```
people1a.csv:2:111111111,Smith,John,111 Main St,,Chicago,IL,66601
```

```
people1a.csv:3:444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304
```

```
people1.csv:2:111111111,Smith,John,111 Main St,,Chicago,IL,66601
```

```
people1.csv:3:222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001
```

- Select a subsequence of records to feed into pipeline:

```
csv-records -hn20 11 personnel.csv | csv-sort 1 > subpersonnel.csv
```

4.20 csv-reformat-date-time

Program to reformat date-time (i.e., timestamp) values in one field of a CSV file.

Usage: `csv-reformat-date-time [OPTION...] FIELD FORMAT_OLD FORMAT_NEW [CSVFILE]`

`FIELD` is the 1-based index of the date-time field to reformat, or header field name if `-Q` option.

`FORMAT_OLD` is the date-time format for existing values.

`FORMAT_NEW` is the date-time format for replacement values.

Options:

- `-b` -- blank records passed through (removed by default)
- `-c` -- take ceiling of seconds (altering minutes) to remove/zero-out seconds
- `-e` -- empty `FIELD` values cause an error (allowed by default)
- `-h` -- header record is to be processed (passed through by default)
- `-Q` -- query header to get index of `FIELD` name
- `-r` -- round seconds (altering minutes) to remove/zero-out seconds
- `-t` -- terminate output records with CR+LF (`\r\n`) (LF only by default)

Date-time `FORMAT`'s must be consistent with `strftime(3)/strptime(3)`:

e.g., `"%D %R"` or `"%m/%d/%y %H:%M"` for `"1/5/21 1:23"` or `"01/05/21 01:23"`;

e.g., `"%m/%d/%Y %T"` or `"%m/%d/%Y %H:%M:%S"` for `"1/5/2021 1:23:45"` or `"01/05/2021 01:23:45"`;

Use `"man strftime"` or `"man strptime"` for further info.

Requires three items be specified:

1. *field* to be reformatted;
2. *format string* for current date-time values;
3. *format string* for new, reformatted date-time values;

The date-time *format strings* define the syntax and meaning of date-time fields. See the section [Date-Time Format Strings](#) for more information. One format string tells how to parse/interpret current values, the other specifies the revised format for the values.

All the date-time values in `FIELD` must use identical syntax, as it can be impossible to unambiguously interpret a date-time string with unknown format (e.g., `2/10/21` could be February 10 2021, or October 2 2021). Date-time values that cannot be parsed with `FORMAT_OLD` will result in an *error*.

Records where `FIELD` is *empty* (empty string) are passed through unchanged by default. Such fields can be made to cause an error with the `-e` option. Records that lack `FIELD` altogether are also passed through unchanged. See [csv-clean](#) and [csv-validate](#) to adjust non-standard CSV files.

If *seconds* appear in the existing `FIELD` values but will be dropped by `FORMAT_NEW`, it may be desirable to apply the *ceiling* or *round* function to update minutes to be a better reflection of the time. This can be done with the `-c` or `-r` options.

If date-time components not in an existing `FIELD` value appear in `FORMAT_NEW`, default values will be used. Default values represent a Linux/UNIX “calendar time” of 0 (zero), what is known as the start of the *UNIX Epoch*: Jan 1, 1970 at midnight. So default times will be: `00:00:00`, default dates will be: January 1, 1970.

Timezone information is *not* required for this program, since it is simply reformatting a date-time, not comparing times or computing time differences.

Sample CSV Files:

- `dates.csv`:

```
Join,Age,Name,Arrive
2019-12-01,34,Bob,1/1/21 00:00:00
2020-01-23,23,Anne,3/17/20 01:34:24
2018-03-04,6,Judy,10/03/19 6:01:49
```

Examples of Using `csv-reformat-date-time`:

- Reformat `date+time` field to ISO format:

```
csv-reformat-date-time 4 '%D %T' '%F %T' dates.csv
```

```
Date,Number,Name,Arrive
12/01/19,34,Bob,2021-01-01 00:00:00
01/23/20,23,Anne,2020-03-17 01:34:24
03/04/18,6,Judy,2019-10-03 06:01:49
```

- Reformat `date+time` field to ISO format, drop seconds:

```
csv-reformat-date-time 4 '%D %T' '%F %R' dates.csv
```

```
Date,Number,Name,Arrive
12/01/19,34,Bob,2021-01-01 00
01/23/20,23,Anne,2020-03-17 01:34
03/04/18,6,Judy,2019-10-03 06:01
```

- Reformat `date+time` field to ISO format, drop seconds using ceiling of seconds:

```
csv-reformat-date-time -c 4 '%D %T' '%F %R' dates.csv
```

```
Date,Number,Name,Arrive
12/01/19,34,Bob,2021-01-01 00:00
01/23/20,23,Anne,2020-03-17 01:35
03/04/18,6,Judy,2019-10-03 06:02
```

- Reformat both “date-time” fields identically:

```
csv-reformat-date-time 1 '%D' '%F %T' dates.csv |
csv-reformat-date-time 4 '%D %T' '%F %T'
```

```
Date,Number,Name,Arrive
2019-12-01 00:00:00,34,Bob,2021-01-01 00:00:00
2020-01-23 00:00:00,23,Anne,2020-03-17 01:34:24
2018-03-04 00:00:00,6,Judy,2019-10-03 06:01:49
```

4.21 csv-remove-duplicate-keys

Program to remove records with duplicate keys in a CSV file. This situation can occur when records for the same item/object/person are updated by adding additional records to a CSV file rather than editing existing records.

Usage: `csv-remove-duplicate-keys [OPTION...] KEYFIELD [CSVFILE]`

KEYFIELD is the 1-based field index for record keys to remove duplicates of, or header field name if `-Q` option.

Options:

- `-f` -- first record of sequence should be left when there are key duplicates (default is to leave last record of sequence with duplicate keys)
- `-h` -- process header (passed through by default)
- `-Q` -- query header to get index of FIELD name
- `-t` -- terminate output records with CR+LF (`\r\n`) (LF only by default)

Note: Duplicate key records must be sequential (grouped), so sorting on KEYFIELD may be required before running this program (stable sort).

The latest record will generally be the first instance or last instance for the item/object/person. This program can remove duplicates for both cases. Removing duplicates can be necessary when trying to merge CSV files (as with [csv-join](#) or [csv-paste](#)).

To use this function, there must be a field in the CSV file with a “key” that identifies each item/object/person considered as the same.

In addition, duplicate key records must occur *sequentially* (i.e., grouped) in the CSV file, so sorting on the key field may be required before running this program. Such a sort must be *stable*, to preserve most recent ordering of original CSV file.

Sample CSV Files:

- `people.csv`:

```
ID,Last_Name,First_Name,Street,Apt,City,State,Zip
111111111,Smith,John,111 Main St,,Chicago,IL,66601
111111111,Smith,John,1300 W. Lake Shore Dr,,Chicago,IL,66601
222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001
333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209
444444444,Harris,Beth,4444 County Rd 34,,Springfield,MO,52304
444444444,Harris,Bethany,4444 County Rd 34,,Springfield,MO,52304
```

Examples of Using `csv-remove-duplicate-keys`:

- Basic call:

```
csv-remove-duplicate-keys 1 people.csv
```

```
ID,Last_Name,First_Name,Street,Apt,City,State,Zip
111111111,Smith,John,1300 W. Lake Shore Dr,,Chicago,IL,66601
222222222,Jones,Mary,22 W. Lake Ave,3A,Carbondale,IL,62001
333333333,Carter,Robert,333 Mountain Rd,,Denver,CO,78209
444444444,Harris,Bethany,4444 County Rd 34,,Springfield,MO,52304
```

4.22 csv-replace-field

Program to *replace/substitute* values in a single field of the records in a CSV file. Basically, “*find and replace*” functionality for a particular field of a CSV file.

Usage: `csv-replace-field [OPTION...] FIELD PATTERN REPLACEMENT [CSVFILE]`

FIELD is the 1-based field index to replace, or header field name if -Q option.

PATTERN is the regex or string pattern to match for replacement.

(PATTERN assumes extended regex syntax, so do not escape regex metachars).

REPLACEMENT specifies the value to replace matched field values with.

Options:

- b -- blank records passed through (removed by default, see also -1)
- F -- interpret PATTERN as a fixed string
- h -- header record is to be processed (passed through by default)
- i -- ignore case when matching PATTERN
- m -- support multiple regex subexpression matches in REPLACEMENT
- o -- omit header record (not processed, not passed through)
- Q -- query header to get index of FIELD name
- t -- terminate output records with CR+LF (\r\n) (LF only by default)
- x -- PATTERN must exactly match the whole field
- 1 -- interpret blank record as having a single empty field (see also -b)

REPLACEMENT can contain '&' to be substituted by matched field substring:

with regex matching '&' replaced by substring matched,

with fixed string matching '&' replaced by entire original field,

-m option requires use of '&0' to match entire regex, and

allows '&1', etc., to match 1st, etc. parenthesize subexpressions.

Requires three items be specified:

1. field to examine and possibly modify;
2. pattern of field value that should be modified;
3. replacement field value (which may include matched value(s) within it);

Can handle *regex* and *fixed string* patterns to denote fields to be replaced (fixed string only when patterns come from a file).

The input CSV file is required to have FIELD in every record, and will terminate with error if missing. *Blank records* are removed by default, but see the -b and -1 options.

Examples of Using `csv-replace-field`:

- Uppcase word “test” if that is field’s value:

```
csv-replace-field -x 'test' 'TEST' file.csv
```

- Uppcase word “test” if it occurs in field’s value:
(note parens required in regex pattern, to delimit pattern components)

```
csv-replace-field -im 2 '(.*)test(.*)' '&1TEST&2' file.csv
```

- Add double quotes around unquoted fields:
(unquoted fields cannot contain embedded special characters, so a valid CSV file will not need further processing to create valid quoted fields)

```
csv-replace-field -x 2 '[^"]*' '"&"' file.csv
```

- Remove any surrounding double quotes from field:
(but note that if there are any embedded special characters, this command will produce an invalid CSV file—must use [csv-process-field](#) instead)

```
csv-replace-field -m 3 '(")(.*)(")' '&2' file.csv
```

- Convert (positive) real number to integer by dropping fractional part after decimal point (i.e., like floor function):

```
csv-replace-field -mx 5 '([0-9]+)(\.[0-9]*)' '&1' file.csv
```

- Add a new field with main category code only, where codes are like Q35.9, and want new field with just Q35:

```
csv-paste file.csv\  
    <(echo "Main_Code";\  
    csv-cat -n file.csv |\  
    csv-cut 7 |\  
    csv-replace-field -mh 1 '([^.]+)(\..+)?' '&1')\  
> file-plus.csv
```

(uses Bash *process substitution* with a pipeline to create desired new field, and `csv-paste` to add it to original file)

4.23 csv-sort

Program to *sort* a CSV file, based on values in a *single field*. [csv-sort-mult](#) is similar but can sort on multiple keys from multiple fields.

Usage: `csv-sort [OPTIONS] SORT_FIELD [CSVFILE]`

`SORT_FIELD` is the 1-based field index to sort on, or header field name if `-Q` option.

Options:

- `-d` -- `SORT_FIELD` is a date-time, use `-D` if need to supply date-time `FORMAT` string
- `-DFORMAT` -- `SORT_FIELD` is a date-time, `FORMAT` the date-time format string
- `-e` -- empty `SORT_FIELD` is an error (allowed by default)
- `-h` -- sort header record (passed through by default)
- `-n` -- numeric sort
- `-o` -- omit header record (not sorted, not passed through)
- `-Q` -- query header to get index of `FIELD` name
- `-r` -- reverse sort
- `-t` -- terminate output records with CR+LF (`\r\n`) (LF only by default)
- `-u` -- unquote quoted `SORT_FIELD` values to obtain sort keys
(remove surrounding `"`'s and unescape embedded `"`'s);
date-time keys are always automatically unquoted
- `-zTIMEZONE` -- timezone for use with date-time sorts, e.g. "America/Chicago"
- `-1` -- interpret blank record as having a single empty field

Date-time `FORMAT` must be consistent with `strptime(3)`:

e.g., `"%m/%d/%Y %T"` (the default) for "1/5/2021 1:23:45" or "01/05/2021 01:23:45";
e.g., `"%D %R"` for "1/5/21 1:23" or "01/05/21 01:23";

Use "man `strptime`" for further info.

`TIMEZONE` required only if `-d/-D` option used and date-times require DST.

The CSV file argument is required to have `SORT_FIELD` in every record. It will terminate with an error if the field is missing in any record. This means that *blank records* are *not* allowed in CSV files input to this program. (Remove with [csv-clean](#) before using this program.)

`SORT_FIELD` is allowed to be *empty* in records by default, but this can be made to cause an error with the `-e` option. Records with empty `SORT_FIELD` values will be sorted to the beginning of the output (or the end with the `-r` option). This will be the case even if `SORT_FIELD` is indicated to be a date-time or number.

The program utilizes the standard Linux/UNIX utility program `sort`, which is run in a subprocess. Several options are thus similar to those of `sort`.

Date-time key values are interpreted using the Linux/UNIX function `strptime(3)`. A *format string* describing the syntax and meaning of date-time fields is required. The default format is given below in the usage message. If this is not appropriate, a custom format string can be supplied using the `-D` option. See the section [Date-Time Format Strings](#) for detailed information. Date-time values that cannot be parsed with a given format string will result in an error.

All date-time values in `FIELD` must use identical syntax, since it can be impossible to unambiguously interpret an unknown date-time string (e.g., 2/10/21 could be February 10 2021, or October 2

2021). Any reasonable CSV file will use a consistent format for date-time values within a single field. Date-time fields can be reformatted with the CSV Utils program [csv-reformat-date-time](#).

Sample CSV Files:

- `people.csv`:

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,Weight_Lbs,Recorded
444444444,Harris,Beth,F,10/30/1987,St. Louis,121,12/13/1987 11:23
333333333,Carter,Robert,M,05/23/2000,Denver,205,02/03/2001 14:02
111111111,Smith,John,M,03/15/1945,Chicago,187,01/15/1985 15:32
222222222,Jones,Mary,F,12/01/1990,Memphis,98,10/24/1991 11:06
666666666,Smith,Anne,F,2/08/2011,Syracuse,68,02/10/2011 15:11
555555555,Smith,Susan,F,06/3/1955,New York,119,04/06/1999 16:05
```

Examples of Using `csv-sort`:

- Sort on text field (note sort is *stable*):

```
csv-sort 2 people.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,Weight_Lbs,Recorded
333333333,Carter,Robert,M,05/23/2000,Denver,205,02/03/2001 14:02
444444444,Harris,Beth,F,10/30/1987,St. Louis,121,12/13/1987 11:23
222222222,Jones,Mary,F,12/01/1990,Memphis,98,10/24/1991 11:06
111111111,Smith,John,M,03/15/1945,Chicago,187,01/15/1985 15:32
666666666,Smith,Anne,F,2/08/2011,Syracuse,68,02/10/2011 15:11
555555555,Smith,Susan,F,06/3/1955,New York,119,04/06/1999 16:05
```

- Sort on numeric field:

```
csv-sort -n 7 people.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,Weight_Lbs,Recorded
666666666,Smith,Anne,F,2/08/2011,Syracuse,68,02/10/2011 15:11
222222222,Jones,Mary,F,12/01/1990,Memphis,98,10/24/1991 11:06
555555555,Smith,Susan,F,06/3/1955,New York,119,04/06/1999 16:05
444444444,Harris,Beth,F,10/30/1987,St. Louis,121,12/13/1987 11:23
111111111,Smith,John,M,03/15/1945,Chicago,187,01/15/1985 15:32
333333333,Carter,Robert,M,05/23/2000,Denver,205,02/03/2001 14:02
```

- *Incorrect* text sort on numeric field:

```
csv-sort 7 people.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,Weight_Lbs,Recorded
555555555,Smith,Susan,F,06/3/1955,New York,119,04/06/1999 16:05
444444444,Harris,Beth,F,10/30/1987,St. Louis,121,12/13/1987 11:23
111111111,Smith,John,M,03/15/1945,Chicago,187,01/15/1985 15:32
333333333,Carter,Robert,M,05/23/2000,Denver,205,02/03/2001 14:02
666666666,Smith,Anne,F,2/08/2011,Syracuse,68,02/10/2011 15:11
222222222,Jones,Mary,F,12/01/1990,Memphis,98,10/24/1991 11:06
```

- Sort on date-time field:

```
csv-sort -d 8 people.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,Weight_Lbs,Recorded
111111111,Smith,John,M,03/15/1945,Chicago,187,01/15/1985 15:32
444444444,Harris,Beth,F,10/30/1987,St. Louis,121,12/13/1987 11:23
222222222,Jones,Mary,F,12/01/1990,Memphis,98,10/24/1991 11:06
555555555,Smith,Susan,F,06/3/1955,New York,119,04/06/1999 16:05
333333333,Carter,Robert,M,05/23/2000,Denver,205,02/03/2001 14:02
666666666,Smith,Anne,F,2/08/2011,Syracuse,68,02/10/2011 15:11
```

- Sort on date-time field, but default FORMAT inconsistent:

```
csv-sort -d 5 people.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,Weight_Lbs,Recorded
Error: record #2: SORT_FIELD was an invalid date-time: '10/30/1987'
```

- Sort on date-time field, supplying FORMAT:

```
csv-sort -D'%m/%d/%Y' 5 people.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,Weight_Lbs,Recorded
111111111,Smith,John,M,03/15/1945,Chicago,187,01/15/1985 15:32
555555555,Smith,Susan,F,06/3/1955,New York,119,04/06/1999 16:05
444444444,Harris,Beth,F,10/30/1987,St. Louis,121,12/13/1987 11:23
222222222,Jones,Mary,F,12/01/1990,Memphis,98,10/24/1991 11:06
333333333,Carter,Robert,M,05/23/2000,Denver,205,02/03/2001 14:02
666666666,Smith,Anne,F,2/08/2011,Syracuse,68,02/10/2011 15:11
```

4.24 csv-sort-mult

Program to *sort* a CSV file, based on values in a *multiple fields*. More flexible than [csv-sort](#) since it can handle multiple keys.

Usage: `csv-sort-mult [OPTIONS] -kKEYDEF... [CSVFILE]`

KEYDEF defines a sort key: `FIELD[n|d][r]`

FIELD is the 1-based index of the key field;

optionally followed by 'n' or 'd' to indicate numeric or date-time sorting for the key;

optionally completed with 'r' to indicate reverse sort;

Examples: `-k1`, `-k2r`, `-k3n`, `-k4nr`, `-k5d`, `-k5dr`.

Options:

`-DFORMAT` -- use FORMAT as the date-time format string for any date-time sort keys

`-e` -- empty SORT_FIELD is an error (allowed by default)

`-h` -- sort header record (passed through by default)

`-o` -- omit header record (not sorted, not passed through)

`-t` -- terminate output records with CR+LF (`\r\n`) (LF only by default)

`-u` -- unquote quoted sort FIELD values to obtain sort keys

(remove surrounding `\''`s and unescape embedded `\''`s);

date-time keys are always automatically unquoted

`-zTIMEZONE` -- timezone for use with date sorts, e.g. `'America/Chicago'`

`-1` -- interpret blank record as having a single empty field

Date-time FORMAT must be consistent with `strptime(3)`:

e.g., `"%m/%d/%Y %T"` (the default) for `"1/5/2021 1:23:45"` or `"01/05/2021 01:23:45"`;

e.g., `"%D %R"` for `"1/5/21 1:23"` or `"01/05/21 01:23"`;

Use `"man strptime"` for further info.

TIMEZONE required only if date-time sort keys require DST.

While this CSV sort program is more flexible than [csv-sort](#), it will be slightly slower and more cumbersome to use when sorting on only a single key. Thus, it is recommended to use [csv-sort](#) for single key sorting.

Syntax is similar to Linux/UNIX `sort` utility, in that each key is specified separately (with `-k` option), and key definition order is the order keys are applied in for sorting.

The CSV file argument is required to have all specified key fields in every record. It will terminate with an error if a key field is missing in any record. This means that *blank records* are *not* allowed in CSV files input to this program. (Remove with [csv-clean](#) before using this program.)

Key fields are allowed to be *empty* by default, but this can be made to cause an error with the `-e` option. Records with empty key field values will be sorted to the beginning of the output sequence based on the key's order (or the end with the `-r` option).

The program utilizes the standard Linux/UNIX utility program `sort`, which is run in a subprocess. Several options are similar to those of `sort`, since `sort` is doing the sorting ultimately.

Date-time key values are interpreted using the Linux/UNIX function `strptime(3)`. A *format string* describing the syntax and meaning of date-time fields is required. The default format is given below

in the usage message. If this is not appropriate, a custom format string can be supplied using the `-D` option. See the section [Date-Time Format Strings](#) for detailed information. Date-time values that cannot be parsed with a given format string will result in an error.

All date-time values in sort key `FIELD`'s must use identical syntax, since only a single format string can be specified and since it can be impossible to unambiguously interpret an unknown date-time string (e.g., 2/10/21 could be February 10 2021, or October 2 2021). Most reasonable CSV files will use an identical format for any date-time values, but if fields originated from difference systems, one might encounter a CSV file where different date-time fields use different formats. Date-time fields in such a file could be made consistent by being reformatted with the CSV Utils program [csv-reformat-date-time](#).

Sample CSV Files:

- `people.csv`:

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,Weight_Lbs,Recorded
4444444444,Harris,Beth,F,10/30/1987,St. Louis,121,12/13/1987 11:23
3333333333,Carter,Robert,M,05/23/2000,Denver,205,02/03/2001 14:02
1111111111,Smith,John,M,03/15/1945,Chicago,187,01/15/1985 15:32
2222222222,Jones,Mary,F,12/01/1990,Memphis,98,10/24/1991 11:06
6666666666,Smith,Anne,F,2/08/2011,Syracuse,68,02/10/2011 15:11
5555555555,Smith,Susan,F,06/3/1955,New York,119,04/06/1999 16:05
```

Examples of Using `csv-sort-mult`:

- Sort on text field (note sort is *stable*):

```
csv-sort-mult -k2 people.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,Weight_Lbs,Recorded
3333333333,Carter,Robert,M,05/23/2000,Denver,205,02/03/2001 14:02
4444444444,Harris,Beth,F,10/30/1987,St. Louis,121,12/13/1987 11:23
2222222222,Jones,Mary,F,12/01/1990,Memphis,98,10/24/1991 11:06
1111111111,Smith,John,M,03/15/1945,Chicago,187,01/15/1985 15:32
6666666666,Smith,Anne,F,2/08/2011,Syracuse,68,02/10/2011 15:11
5555555555,Smith,Susan,F,06/3/1955,New York,119,04/06/1999 16:05
```

- Sort on two text fields:

```
csv-sort-mult -k2 -k3 people.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,Weight_Lbs,Recorded
3333333333,Carter,Robert,M,05/23/2000,Denver,205,02/03/2001 14:02
4444444444,Harris,Beth,F,10/30/1987,St. Louis,121,12/13/1987 11:23
2222222222,Jones,Mary,F,12/01/1990,Memphis,98,10/24/1991 11:06
6666666666,Smith,Anne,F,2/08/2011,Syracuse,68,02/10/2011 15:11
1111111111,Smith,John,M,03/15/1945,Chicago,187,01/15/1985 15:32
5555555555,Smith,Susan,F,06/3/1955,New York,119,04/06/1999 16:05
```

- Sort on numeric field:

```
csv-sort-mult -k7n people.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,Weight_Lbs,Recorded
6666666666,Smith,Anne,F,2/08/2011,Syracuse,68,02/10/2011 15:11
2222222222,Jones,Mary,F,12/01/1990,Memphis,98,10/24/1991 11:06
5555555555,Smith,Susan,F,06/3/1955,New York,119,04/06/1999 16:05
4444444444,Harris,Beth,F,10/30/1987,St. Louis,121,12/13/1987 11:23
1111111111,Smith,John,M,03/15/1945,Chicago,187,01/15/1985 15:32
3333333333,Carter,Robert,M,05/23/2000,Denver,205,02/03/2001 14:02
```


- *Incorrectly* sort on numeric field without n modifier:

```
csv-sort-mult -k7 people.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,Weight_Lbs,Recorded
555555555,Smith,Susan,F,06/3/1955,New York,119,04/06/1999 16:05
444444444,Harris,Beth,F,10/30/1987,St. Louis,121,12/13/1987 11:23
111111111,Smith,John,M,03/15/1945,Chicago,187,01/15/1985 15:32
333333333,Carter,Robert,M,05/23/2000,Denver,205,02/03/2001 14:02
666666666,Smith,Anne,F,2/08/2011,Syracuse,68,02/10/2011 15:11
222222222,Jones,Mary,F,12/01/1990,Memphis,98,10/24/1991 11:06
```

- Sort on date-time field:

```
csv-sort-mult -k8d people.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,Weight_Lbs,Recorded
111111111,Smith,John,M,03/15/1945,Chicago,187,01/15/1985 15:32
444444444,Harris,Beth,F,10/30/1987,St. Louis,121,12/13/1987 11:23
222222222,Jones,Mary,F,12/01/1990,Memphis,98,10/24/1991 11:06
555555555,Smith,Susan,F,06/3/1955,New York,119,04/06/1999 16:05
333333333,Carter,Robert,M,05/23/2000,Denver,205,02/03/2001 14:02
666666666,Smith,Anne,F,2/08/2011,Syracuse,68,02/10/2011 15:11
```

- Sort on date-time field, but default FORMAT inconsistent:

```
csv-sort-mult -k5d people.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,Weight_Lbs,Recorded
Error: record #2: KEYDEF field 5 was an invalid date-time: '10/30/1987'
```

- Sort on date-time field, supplying FORMAT:

```
csv-sort-mult -k5d -D'%d/%m/%Y' people.csv
```

```
ID,Last_Name,First_Name,Sex,Birthdate,Birth City,Weight_Lbs,Recorded
111111111,Smith,John,M,03/15/1945,Chicago,187,01/15/1985 15:32
555555555,Smith,Susan,F,06/3/1955,New York,119,04/06/1999 16:05
444444444,Harris,Beth,F,10/30/1987,St. Louis,121,12/13/1987 11:23
222222222,Jones,Mary,F,12/01/1990,Memphis,98,10/24/1991 11:06
333333333,Carter,Robert,M,05/23/2000,Denver,205,02/03/2001 14:02
666666666,Smith,Anne,F,2/08/2011,Syracuse,68,02/10/2011 15:11
```

4.25 csv-validate

Program to validate that a CSV file is properly formatted according to RFC 4180, plus provide additional info about the file's "style" characteristics as a CSV. See examples below for info that is provided.

Usage: `csv-validate [CSVFILE]`

Options:

`-v -- verbose: print messages about each issue encountered`

Verbose syntax/style messages and CSV characteristics report are output to standard output. Error/warning messages are to standard error.

Example Output from csv-validate:

- CSV file with quoted fields and embeds:

Report for 'embeds.csv':

```
-----  
Basics:  
  Records in total:                      5  
  Fields in header record:                5  
Syntax:  
  Records with field count mismatches:    0  
                                           short:    0  
                                           extra:    0  
  Blank records:                         0  
Style:  
  Fields with embedded carriage-returns:  2  
  Embedded carriage-returns:              3  
  Fields with embedded linefeeds:         2  
  Embedded linefeeds:                    3  
  Quoted fields:                         25  
  Cleanable quoted fields:                21  
  Unquoted empty fields:                  0  
  Unquoted whitespace fields:            1  
  Unquoted leading/trailing whitespace fields: 4
```

- CSV file with non-uniform field counts:

Report for 'notuniform.csv':

```
-----  
Basics:  
  Records in total:                      7  
  Fields in header record:                5  
Syntax:  
  Records with field count mismatches:    4  
                                           short:    2  
                                           extra:    2  
  Blank records:                         0  
Style:  
  Fields with embedded carriage-returns:  0  
  Embedded carriage-returns:              0  
  Fields with embedded linefeeds:         0  
  Embedded linefeeds:                    0  
  Quoted fields:                         0  
  Cleanable quoted fields:                0  
  Unquoted empty fields:                  0  
  Unquoted whitespace fields:            1  
  Unquoted leading/trailing whitespace fields: 4
```

4.26 csv-to-dsv

Program to convert a CSV file to a DSV (*delimiter-separated values*) file that uses a non-comma delimiter char. Changes delimiter from comma (,) to specified delimiter char.

Usage: `csv-to-dsv [OPTION...] DSV_DELIMITER_CHAR [CSVFILE]`

(DSV_DELIMITER_CHAR can use C-style escape-sequence if quoted: e.g., `"\t"`)

Options:

- b -- blank records passed through (removed by default)
- r -- replace embedded CR (`\r`) and LF (`\n`) characters with text
- t -- terminate output records with CR+LF (`\r\n`) (LF only by default)
- u -- unquote quoted fields that can be unquoted with delimiter change

Automatically quotes fields if necessary, using *dquotes* (`'`'s) and escaping embedded dquotes.

Output is a DSV file that should be valid if quoting with dquotes is allowed (different DSV standards may handle field “quoting” differently).

Examples of Using `csv-to-dsv`:

- “Tabify” CSV file (C-style tab):
`csv-to-dsv "\t" example.csv > example.tsv`
- “Tabify” CSV file (Bash tab):
`csv-to-dsv $'\t' example.csv > example.tsv`
- Replace commas, so may allow unquoting of fields with embedded commas:
`csv-to-dsv -u ":" example.csv > example.dsv`

4.27 dsv-to-csv

Program to convert DSV (*delimiter*-separated values) file that uses a non-comma delimiter char, to a valid CSV file. Changes delimiter from specified delimiter to comma (,).

Usage: dsv-to-csv [OPTION...] DSV_DELIMITER_CHAR [DSVFILE]

(DSV_DELIMITER_CHAR can use C-style escape-sequence if quoted: e.g., "\t")

Options:

- b -- blank records passed through (removed by default)
- r -- replace embedded CR (\r) and LF (\n) characters with text
- t -- terminate output records with CR+LF (\r\n) (LF only by default)
- u -- unquote quoted fields that can be unquoted with delimiter change

Automatically quotes fields if necessary (consistent with CSV standard).

Output will be a valid CSV file if DSV file uses dquotes (' ' 's) for quoting. Different DSV standards may handle field “quoting” differently, and if that is the case, then the result is not guaranteed to be valid CSV syntax.

Examples of Using dsv-to-csv:

- Convert TSV to CSV file (C-style tab):

```
dsv-to-csv "\t" example.tsv > example.csv
```

- Convert TSV to CSV file (Bash tab):

```
dsv-to-csv $'\t' example.tsv > example.csv
```

5 CSV File Syntax

The main standard for CSV files is *RFC 4180*, though it is possible to find “CSV files” that do not strictly adhere to RFC 4180. All CSV Utils component programs are able to handle CSV files compatible with RFC 4180. This includes *quoted fields* containing *embedded special characters* (see below).

Summary of CSV syntax according to RFC 4180:

Terminology and notation:

- *special characters*:
 - comma (,), ASCII 0x2C
 - double-quote ("), ASCII 0x22
 - carriage-return (\r), ASCII 0x0D, denoted **CR** below
 - linefeed (\n), ASCII 0x0A, denoted **LF** below
- *text characters*:
 - all the *printable* ASCII characters, 0x20 through 0x7E (*space* through *~*), excluding the *special characters* comma (,) and double-quote (")
- **CRLF** denotes the two-character sequence carriage-return + linefeed (\r\n)

CSV file syntax:

- a *CSV file* is made up of one or more *records*, records separated by **CRLF**
- the final record in a CSV file need not be followed by a **CRLF** terminator
- a *record* is made up of one or more *fields*, fields separated by a single *comma*
- all records must contain the same number of fields
- each *field* may be either *unquoted* or *quoted*
- *unquoted* fields consist of zero or more *text characters*
- *quoted* fields consist of:
 1. a *double-quote* (")
 2. zero or more *text characters* and *special characters*
 3. a *double-quote* (")
- every *double-quote* character (") that is embedded within a *quoted field* must be *escaped*: converted to a sequence of *two* double-quote characters (")
- both unquoted and quoted fields may be *empty* (have value of “the empty string”)

Headers:

- the *first record* in a CSV file may be a *header*
- the field values in a header are considered as the *names* of the fields in the CSV file
- header fields must be valid fields as defined above
- CSV files are not required to have a header

Detailed points on RFC 4180 syntax and the CSV Utils:

- RFC 4180 specifies CSV file record terminators/separators as the CRLF character sequence (i.e., C string "\r\n"). However, it is more common (and preferred) on Linux/UNIX systems to use only single LF (C char '\n') characters as record terminators/separators—just as is done with text file *lines*. The CSV Utils programs are able to handle either LF only or CRLF record terminators when reading CSV files. By default, the CSV Utils programs *use LF only record terminators for CSV output*. However, the programs each include an option for outputting CRLF terminators instead.
- “Blank records” are records that are *empty*: they contain no text characters or comma field separators. I.e., these are records that would commonly be called *blank lines* in text files, and would appear as blank lines if a CSV file was opened in a text editor. (Note, though, that “blank lines” in text files might also contain only whitespace, but records consisting of whitespace are not “blank records,” since they would be interpreted as containing a single unquoted field.)

RFC 4180 specifies that records with *no fields* are *not legal* CSV files. However, it is ambiguous whether a blank record in a CSV file represents an illegal blank (no fields) record or a record containing *one empty unquoted field*. Of course, this ambiguity should be an issue only for “CSV files” consisting of a single field (and such files are questionable as being CSV files). See Section 6.4 for more information about how the CSV Utils programs handle “blank” records.

- Records with *non-uniform* numbers of fields are not legal under RFC 4180. Treatment of “CSV files” where records contain non-uniform numbers of fields varies among the CSV Utils programs based on their functionality. Read the documentation carefully if operating on CSV files containing records with non-uniform numbers of fields.
- RFC 4180 is ambiguous on the treatment of spurious *leading/trailing whitespace* (i.e., leading/trailing whitespace that is *not* part of field values). Spurious whitespace is *not* allowed by the CSV Utils. In unquoted fields, it will be included as part of a field’s value. Whitespace before/after the quotes in a quoted field will be considered a CSV syntax error.
- RFC 4180 addresses only ASCII characters, but modern CSV files may use alternative character sets. The CSV Utils programs use C `char` types, i.e., handle only 8-bit/byte characters. However, any character encoding that is *backward compatible* with ASCII can be correctly parsed with the CSV Utils, because the CSV *special characters* can be correctly located and interpreted by considering only individual bytes. This means that 8-bit ASCII extensions like *Windows-1252/ISO-8859-1* will work fine, as will byte-based but variable-length encodings like *UTF-8*. By contrast, UTF-16, which uses 16-bit units, is *incompatible*! Use a utility like `iconv` to convert such files to a compatible encoding, such as UTF-8:
`iconv -f UTF-16LE -t UTF-8 infile > outfile`

DSV Files:

- CSV files are one type of *Delimiter Separated Values* (*DSV*) file format. DSV files consist of a sequence of *records*, each consisting of *fields* that are separated by a single *delimiter* character. Besides the comma used in CSV files, other common delimiters are the tab character, colon (:), vertical bar (|), and space character. Unfortunately, many of these alternative DSV formats are less well formalized and/or less flexible than the RFC 4180 CSV format.
- For example, there is an IANA standard for *Tab Separated Values* (*TSV*) files, but that standard is extremely simple and inflexible: it does not allow embedded tab characters or carriage-returns or linefeeds, nor empty fields. Because of these limitations, TSV files are often found that use various approaches to deal with these limitations. For example, embeds may be *escaped*: `\n` (linefeed), `\r` (carriage-return), `\t` (tab), `\\` (backslash). Alternatively, the double quoting approach from CSV files may be used for embeds.
- The CSV Utils programs `csv-to-dsv` and `dsv-to-csv` can be used to convert to/from alternative delimiter DSV files. However, these utilities follow the CSV approach for handling embeds: double-quoting fields.

6 General Behavior Notes

6.1 Header Records

Default behavior is intended to be most likely to produce a valid CSV file. So, header records will be printed by default. Thus, `csv-grep` will output the header even if no records are matched (check exit status to determine if any matches, as with standard `grep`).

Many CSV Utils programs accept options for different behavior:

- `-h` to process the header (i.e., for CSV files w/o a header)
- `-o` to omit the header from the output

6.2 File Arguments and Standard Input

All CSV Utils except for `csv-edit` and `csv-get-*` do not require file arguments, reading from *standard input* otherwise. `csv-edit` requires a CSV file argument specified, since it modifies the file. The shell script-specific programs `csv-get-field` and `csv-get-record` read from *standard input only*, so they do not take file arguments.

Most CSV Utils programs take at most a *single* CSV file argument.

The CSV Utils programs that can accept *multiple file args* are:

`csv-cat`, `csv-counts-records`, `csv-print`.

6.3 Empty “CSV” Files

Technically, an *empty file* (zero bytes in size) cannot be considered a valid CSV file. Nonetheless, most of the CSV Utils programs will treat an empty file argument (or stdin stream) as a CSV file with zero records, and will simply terminate immediately upon trying to read the header/first record. In particular: there will *not be any warning/error message*. E.g., `csv-grep` will output nothing at all (though its exit status will be 1/failure).

The purpose of this approach is to allow for pipelines where some stage results in no records, without having warning/errors from subsequent stage(s). For example, suppose you have two `csv-grep` stages on different fields, and for some input file, they result in zero records passing through (and headers are not passed). One does not generally desire getting error messages from all later stages. Rather, getting empty output indicates the stages filtered all records out.

The exceptions to this behavior—i.e., CSV Utils programs that produce warning/error messages with empty files—are those that:

- operate on specific ranges of records: `csv-print`, `csv-records`
- “merge” two CSV files: `csv-join`, `csv-paste`
- evaluate/fix files: `csv-reformat-date-time`, `csv-validate`
- must have a header record to operate: `csv-names2nums`

If a program is asked to print particular range of records and there are no such records, it seems reasonable to print a warning rather than just nothing. Likewise if asked for a certain number of records, and there are not that many. “Merge” operations require two files, so having any empty file makes any sort of merge quite impossible.

6.4 Blank (Empty) Records

As noted in Section 5, “blank” records (“records” that consist of no characters/fields) are *not* legal under RFC 4180. However, there are two key issues with “blank” records that were considered for the CSV Utils programs handling of them:

- a single field record, where the field value is empty (and unquoted) is indistinguishable from a truly *blank* record;
- blank records/lines can be useful to improve readability of spreadsheets;

Blank records might reasonably be assumed to be a single empty field when the CSV file has only a single field. Otherwise, assuming this would make the CSV file invalid due to creating nonuniform numbers of fields.

Blank records can be useful as separators in CSV files that are intended to be viewed in spreadsheet programs. Blank records could also result from stages in pipelines.

When a blank record is being assumed to be a blank, a CSV program could handle it as:

1. pass it through without attempting to process it;
2. process it if reasonable (i.e., if specific field not required);
3. remove it (i.e., don’t process it or pass it through);
4. throw an error;

The general approach that the CSV Utils programs take to “blank” records is to *remove them* by default. I.e., they are not processed and not passed through. However, because of the above noted considerations, many of the CSV utils programs can accept one or both of the following options, which modify their behavior with regard to blank records:

- **-b** – pass blank records through (unprocessed);
- **-1** – interpret blank records as single empty field records;

Read the documentation for a program if operating on CSV files containing “blank” records. If an option is not available for a program, it is because it makes no sense. E.g., passing blank records through cannot reasonably be done for **csv-sort**.

7 Integrating CSV Utils with Standard Linux Utilities

Many of the standard Linux/UNIX utilities assume that files consist of linefeed-terminated (`\n`) lines of text. For example: `AWK`, `cut`, `grep`, `sed`, `sort`, `tail`, `wc`, etc. While these utilities can be applied to many CSV files, valid CSV files can include *embedded linefeeds*—i.e., linefeeds inside of *quoted fields* (see [CSV File Syntax](#) for more info). CSV files that contain embedded linefeeds will generally cause CSV records to be *incorrectly* processed by the standard Linux/UNIX utilities mentioned above.

While CSV Utils programs can replace much of the functionality one might want from standard Linux/UNIX utilities, it could still be desirable to be able to apply standard utilities from time to time. Since the CSV Utils programs were designed to be easily used in pipelines, it is often possible to fix issues with the standard utilities by integrating CSV Utils programs into a pipeline along with the standard utilities.

A number of GNU versions of standard Linux/UNIX utilities have an option that will interpret “lines” as being terminated by *null chars* (`\0`’s) instead of linefeeds (i.e., “*zero terminated*” files). The GNU utilities that are known to have this capability are: `comm`, `cut`, `head`, `join`, `numfmt`, `paste`, `sed`, `shuf`, `sort`, `tail`, `uniq`.

Several CSV Utils programs have an option that will cause output records to be *terminated by null chars* (instead of linefeeds). These programs can then be used to start a pipeline that feeds into standard utilities, allowing those utilities to deal with CSV files containing embedded linefeeds. The CSV Utils programs with this capability are: `csv-cat`, `csv-cut`, `csv-get-record`, `csv-records`.

Here is an example pipeline, that uses `tail` to print the last 5 records in a CSV file, even if the CSV file contains embedded linefeeds:

```
csv-cat -z embeds.csv | tail -z -n5 | null2lf
```

- `-z` option is used with `csv-cat` to null-char-terminate CSV records;
- `-z` option is used with `tail` to see null-char-terminated records as lines;
- script `null2lf` is used last to convert the null chars to linefeeds;

`null2lf` is a simple one-line *shell script* provided with the CSV Utils programs, which can be used to convert “zero-terminated” files back to valid (linefeed-terminated) CSV records. It runs `tr` to convert null-char’s (`\0`’s) to linefeeds (`\n`’s).

GNU AWK can also be used with CSV files containing embedded linefeeds, but a bit more effort is required. By default, AWK breaks files into “input records” on linefeeds. However, it is possible to change AWK’s *record separator* (`RS`), so files are broken into records on null chars. CSV Utils programs can then be used to feed AWK CSV records that will be properly parsed.

Here is an example pipeline, with `csv-cat` feeding GNU AWK code that simply outputs CSV records prefixed with a record number:

```
csv-cat -z embeds.csv | awk -v RS="\0" -v OFS="," ' {print NR,$0} '
```

- record separator (`RS`) is changed to null char;
- *output record separator* (`ORS`) is `"\n"` by default;
- but *output field separator* (`OFS`) default is spaces, so it must be changed to CSV format;

Note that this AWK issue *does not apply* to the CSV Utils programs that use AWK code for processing logic (`csv-grep-cond`, `csv-grep-cond-records`, `csv-process-field`, `csv-process-records`). Those programs automatically make required modifications to their (internal) AWK invocations.

8 AWK Code Basics

Several CSV Utils programs make use of **AWK** to provide the ability to supply program code to manipulate or evaluate CSV files: **csv-grep-cond**, **csv-grep-cond-records**, **csv-process-field**, and **csv-process-records**.

These programs utilize the system-installed AWK, run in a subprocess. The system AWK for Linux is assumed to be GNU AWK. Alternative AWK versions may not be compatible with the CSV Utils programs. The GNU AWK manual can be found at:

<https://www.gnu.org/software/gawk/manual/gawk.html>

Pay attention to the *AWK Coding Requirements* information in the manual sections for each of the relevant CSV Utils programs, as failure to follow these requirements can result in unwanted behavior.

In general, AWK code used with the CSV Utils should:

- Not modify `RS`, `FS`, `ORS`, `OFS`, `FPAT`.
- Not call `print` or `printf`.
- Not call `next` or `nextfile`.

Invalid AWK code passed to any of the CSV Utils programs will generally result in error messages from the AWK subprocess, and termination of the programs. Fix the coding errors and rerun.

8.1 AWK Expressions vs Programs

Two of the CSV Utils programs that make use of AWK code, require AWK *expressions*: **csv-grep-cond** and **csv-grep-cond-records**.

The other two CSV Utils programs that make use of AWK code, require AWK *programs*: **csv-process-field** and **csv-process-records**.

An AWK *expression*, is AWK code that evaluates to (returns) a *value*. I.e., it is what could be used in an `if` or `while` condition. For example, “`$0>=10`”, which evaluates to 1 or 0 (for true or false)

An AWK *program*, must be a syntactically valid AWK program, and can perform arbitrary manipulation of the CSV record information passed to it. See **csv-process-field** and **csv-process-records** to understand how record information is passed, and how records can be modified by making assignments to `$0`, `$1`, etc.

An AWK program consists of a set of *rules*, where each rule consists of a (optional) *pattern* and an *action*. It is expected that AWK programs used with **csv-process-field** and **csv-process-records** will be fairly simple, similar to the kinds of AWK “programs” typically used with shell scripts. Thus, the program code may consist of a single rule, without a pattern, so just an action. Each action is enclosed in *curly braces* (`{}`’s). So AWK program code used with CSV Utils is often going to be of the form: “`{action}`”, where *action* involves an assignment. E.g., “`{ $0=substr($0,1,2) }`” or “`{ if ($1!="") $2=$1 }`”.

A general description of AWK program syntax is beyond the scope of this manual. See the GNU AWK manual: <https://www.gnu.org/software/gawk/manual/gawk.html>

AWK code should generally be *quoted* when supplied to CSV Utils, to avoid issues with the shell interpreting AWK code characters as shell metacharacters. Using Bash single quotes to surround AWK code simplifies the use of double quotes for strings: e.g., `'$0=="Test"'` or `'{ $0=substr($0,1,3) }'`

8.2 Assignments to \$0

With **csv-process-field**, the new value for `FIELD` in each CSV record will be the value of `$0` after the AWK code runs on the CSV record. The main way to change the value of `FIELD` in CSV records is by having the AWK code assign a new value to `$0`: e.g. `$0=newvalue`. One tricky aspect of this is that assignments to `$0` *must be done with string values!* Typically, AWK automatically converts back and forth between strings and numbers, as required. This is not the case with an assignment to `$0` however. Assigning a number to `$0` results in `$0` becoming the *empty string*. E.g., code like `$0=int($0+0.5)` will result in `$0` having an empty string value.

An easy way to ensure `$0` gets a string value even if being assigned from what AWK considers a number, is to always concatenate the value with the empty string in the assignment to `$0`: `$0=""int($0+0.5)` Another approach around this problem is to assign to `$1` (field 1) instead of `$0`, since then number to string conversion happens as normal. (Do not do assignments to fields > 1 , as this will cause fields to be added to the CSV records.) One can also use `sprint()` (see below), to make sure a numeric value is a string for `$0`. `sprint()` also allows one to have control over the numeric conversion.

8.3 Formatting Real Numbers

AWK automatically converts numbers to strings using a default format (held in `CONVFMT`). The more general approach for outputting numeric values is to use the `sprintf()` function: `$0=sprintf("%.2f",$0+0.5)`.

Using `sprintf()` allows one to have greater control over the format of resulting numbers. In particular, one can control the *precision* (number of decimal points) that get used with real numbers. One can also use `sprintf()` to make sure a numeric value is a string for `$0` (see above).

8.4 Alternative Cases in AWK Code

If different processing actions need to be taken depending on e.g. `FIELD`'s value, AWK code needs to be structured to ensure that only the correct code case always gets run. There are two approaches that can be used:

- a single rule containing *if-then-else-if-else* logic
- multiple rules with *non-overlapping patterns*

For example, consider processing a field that could include (positive) integers or reals, where we want to leave integer values unchanged, but want to *round* real values to integers.

One approach to rounding only reals would be to use a rule without a pattern, and have appropriate *if-then* logic within the rule's action:

```
{if ($0 ~ /^[0-9]+\.[0-9]+$/) {$0=sprintf("%d",int($0+0.5))}}
```

Another approach would be to use a rule pattern so that the action is applied only to reals: `/^[0-9]+\.[0-9]+$/{$0=sprintf("%d",int($0+0.5))}`

With both examples, only real values are modified, because the regex will not match integers. In the first case, the rule will get fired on all records, but modify only those with real values. In the second case, the rule will get fired only with reals, so integers will not be affected, because no code will be run.

When using multiple AWK *rules* that have different *regex patterns*, one will typically want to ensure that only a single rule applies to each field (since AWK will apply all rules that match a field value). Ensuring that only a single rule’s pattern matches can sometimes be tricky, since there are limited regex “negation” operators. One can however, negate an entire rule pattern: `!/^[0-9]+\.[0-9]+$/{...}`.

This means that having just two alternative rules is generally easy:

```
# Rule to process reals:
/^[0-9]+\.[0-9]+$/{
    ...
}

# Rule to process all else:
!/^[0-9]+\.[0-9]+$/{
    ...
}
```

If multiple, alternative code cases are required, it will probably be simpler to structure the AWK code as a *single rule* that applies to all fields, and use appropriate *if-then-else-if-else* logic within that rule to ensure that only a single case applies to each FIELD value:

```
{
    intval = int($0)
    frac = $0 - intval
    if (frac == 0)
        $0 = intval
    else if (frac >= 0.5)
        $0 = intval + 1
    else #frac < 0.5
        $0 = intval
}
```

Note that the above code can be written on a single line by using `;`’s:

```
{intval=int($0);frac=$0-intval;if(frac==0)$0=intval;else if(frac>=0.5)$0=intval+1;...}
```

However, since this is rather long and confusing to read, it is probably better committed to a file.

8.5 AWK getline

The CSV Utils that make use of AWK, set AWK’s RS and FS parameters to ensure AWK correctly interfaces with the CSV Utils pipelines. This can result in AWK’s *getline not working as it typically does*. However, this should have little practical effect, as *getline* is an advanced feature of AWK. The GNU AWK manual even says this: “The *getline* command is used in several different ways and should not be used by beginners.”

The only use of *getline* that is compatible with the CSV Utils is to run a command/program that produces a *single line of output* that must be captured (as a string). Using *getline* to read lines from files is *absolutely not* compatible with the CSV Utils!

Here is a function that can run a command-line command/program and capture its (single line of) output, and is compatible with the CSV Utils:

```
# Function to run a command-line command and return its output.
# Returns -1 on error from command.
function runcmd(cmd, cmdwredir,value,status)
{
    cmdwredir = cmd" 2>/dev/null"
    status = cmdwredir | getline value
    close(cmdwredir)

    #Check for command failure and return -1:
    if (status != 1) return -1

    #Remove trailing linefeed when RS != \n:
    sub("\n","",value)

    return value
}
```

The above function can be used, for example, to run the `date` command as follows:

```
# Function to run a the date command-line command and return its output.
# Returns "" (empty string) on error running date command.
function rundate (datetime,format,timezone, dateout)
{
    dateout = runcmd("TZ="timezone " date -d\" datetime \" \" format \"\")
    if (dateout == -1) return ""
    return dateout
}
```

9 Date-Time Format Strings

Several CSV Utils programs make use of *format strings* to interpret or format *date-time* values in CSV file fields: [csv-reformat-date-time](#), [csv-sort](#), [csv-sort-mult](#). Date-time format strings have to be consistent with the Linux/UNIX functions `strftime(3)` and `strptime(3)`. See “`man strftime`” or “`man strptime`” for complete information.

Here are the most frequently used format string directives for individual date-time items:

```
%a      The name of the day of the week, abbreviated.
%A      The name of the day of the week, full.
%b      The month name, abbreviated.
%B      The month name, full.
%d      The day of the month (range 01 to 31).
%H      The 24-hour clock hour (0-23).
%I      The 12-hour clock hour (1-12).
%m      The month number (1-12).
%M      The minute (0-59).
%p      AM or PM.
%P      am or pm.
%S      The second (0-60; 60 may occur for leap seconds).
%y      The year without a century (range 00 to 99).
%Y      The year including the century (e.g., 1998).
```

There are also a few format string directives for combination date-time items:

```
%D      American style date, equivalent to: %m/%d/%y.
%F      ISO style date, equivalent to: %Y-%m-%d.
%r      12-hour clock time w/seconds, equivalent to: %I:%M:%S %p.
%R      24-hour clock time w/o seconds, equivalent to: %H:%M.
%T      24-hour clock time w/seconds, equivalent to %H:%M:%S.
```

Here are a few common example date-time values and appropriate format strings:
(xx/xx/xx is “American” style date: month/day-of-month/year)

Date-Time	Format String	Alternative Format
1/5/21 3:47	“%m/%d/%y %H:%M”	“%D %R”
01/05/21 3:47	“%m/%d/%y %H:%M”	“%D %R”
01/05/21 3:47:32	“%m/%d/%y %H:%M:%S”	“%D %T”
1/5/2021 3:47	“%m/%d/%Y %H:%M”	“%m/%d/%Y %R”
01/05/2021 3:47	“%m/%d/%Y %H:%M”	“%m/%d/%Y %R”
01/05/2021 3:47:32	“%m/%d/%Y %H:%M:%S”	“%m/%d/%Y %T”
2021-01-05 3:47	“%Y-%m-%d %H:%M”	“%F %R”
2021-01-05 3:47:32	“%Y-%m-%d %H:%M:%S”	“%F %T”

Note how separator characters like /, :, -, and whitespace must be appropriately included along with the date-time directives in format strings.

Important note about %y and %D:

The %y and %D formats can lead to *misinterpretation of dates*, so must be treated very carefully!

Glibc currently interprets two-digit year values as follows:

- 69 to 99: 1969 to 1999
- 0 to 68: 2000 to 2068

Thus, if dates in CSV files extend from before 1969 to after 1969, two-digit years will be incorrectly interpreted, so must be converted to four-digit format. [csv-replace-field](#) could be used to do this, taking into account what year move from the 1900's to the 2000's.

10 Build-Time Parameters

The header file `libcsv.h` contains a number of *preprocessor constants* that control various aspects of the CSV Utils programs when they are built (compiled). This section describes these parameters, in case changes are necessary. Note that after any changes are made, the CSV Utils programs will have to be recompiled.

10.1 CSV File Size Limits

Since a key goal of these utilities is speed, fixed-size memory blocks are used when CSV records or fields must be stored. While the fixed-sizes should be sufficient for virtually all CSV files, if extremely large CSV files need to be processed, memory sizes may have to be increased. Alternatively, if the settings are known to be much larger than necessary, sizes could be reduced to reduce process sizes and slightly improve performance. Note that CSV Utils programs will throw an error if a limit is exceeded!

The CSV file size preprocessor constants along with their default values, are:

- **MAX_FIELDS:** 512
The maximum *number of fields* allowed in each CSV record.
An array to hold fields, `char* fields[MAX_FIELDS]`, will be one page: $512 \times 8B = 4096B$.
- **MAX_RECORD_SIZE:** 65536
The maximum *size* (in bytes) allowed for each CSV *record*.
An array to hold records, `char record[MAX_RECORD_SIZE]`, will be 16 pages (same as pipe buffer size).
- **MAX_FIELD_SIZE:** 8192
The maximum *field size* (in bytes) allowed for functions that return a field.
An array to hold fields, `char field[MAX_FIELD_SIZE]`, will be 2 pages.

Note that *no limit* is imposed on the *number of records* that can be in CSV files.

10.2 Embedded Characters Replacements

Some CSV files contain *carriage-return* (`\r`) and *linefeed* (`\n`) characters *embedded* in *quoted* fields. These are valid CSV files in line with RFC 4180, and the CSV Utils programs can handle such files. However, embedded CR/LF characters will make it impossible to use most standard Linux/UNIX text file utilities (e.g., `grep`, `sort`, `wc`).

The CSV Utils program `csv-clean` can *remove embedded CR/LF characters*, replacing them with *text*. This can be useful because it will allow those CSV files to be processed using standard Linux/UNIX text file utilities, as well as allowing them to be more easily viewed as text.

Four preprocessor constants control what text is used when embedded CR/LF characters are replaced. The default values make it quite easy to view fields as text. The length of (numbers of characters in) the replacement text sequences must be provided as well, to eliminate the costs of repeatedly having to determine the replacement string lengths in code.

While the default replacements are easy to read, a common approach with *TSV* files is to replace the characters with their C language (two character) escape sequences: `\r` and `\n`. It should be easy to substitute alternative strings using the `sed` utility in a pipeline, if desired.

The four relevant preprocessor constants and their default values are:

- **CR_REPLACEMENT**: "<CR>"
The replacement text for embedded CR's (\r's).
- **CR_LENGTH**: 4
The length of **CR_REPLACEMENT**.
- **LF_REPLACEMENT**: "<LF>"
The replacement text for embedded LF's (\n's).
- **LF_LENGTH**: 4
The length of **LF_REPLACEMENT**.

10.3 Utility Program Paths

Several CSV Utils programs make use of standard Linux/UNIX utility programs run in subprocesses and communicating via pipelines. These utility programs are: **awk** and **sort**. While the paths for these programs could be found automatically via a user's **PATH** (e.g., using **execvp()**), this can involve some overhead from trying each **PATH** component. Since, again, speed is a key goal of these CSV Utils programs, the standard paths for these utility programs are provided via preprocessor constants. This allows the CSV Utils programs to avoid having to search (**execv()** is used instead of **execvp()**).

Should **awk** or **sort** be installed in non-standard locations, the following preprocessor constants will need adjusting:

- **AWK_PATH**: "/usr/bin/awk"
- **SORT_PATH**: "/usr/bin/sort"

11 Performance

The CSV Utils component programs are all written in C for efficiency, and also generally written in a style that emphasizes execution speed. In some cases, the emphasis on speed has meant there is some “code duplication” to avoid repeated options testing on every field or even in every record. While this might slightly hamper maintenance programming, it is intentional, having had its value confirmed by testing on large CSV files.

Performance testing was done on a machine with an Intel Core i7-6700 CPU @3.40GHz. A CSV file with a header plus one million records was used for testing. Each record had 10 fields. Total file size was 161MB, with records on average 180 bytes. The test files were always cached due to repeated tests and 32GB of RAM.

The goal of this testing was to establish base speeds of the programs, not e.g., average run time on a particular machine with a particular load. After some testing, it was decided to carry out four sets of 25 program runs (five sets of 10 runs with slower programs), and report the average of the *minimum* elapsed times from each of the run sets, taking the ceiling of the average (to the next full hundredth/tenth of a second). This was found to be very consistent, and should be representative of the inherent speeds of these programs (unlike, e.g. mean run times, which vary with machine load).

Average minimum elapsed time results for select CSV Utils:

- `csv-counts-records test.csv`
real time: 0.57 sec (time to read/parse CSV file and count records)
- `csv-records 1 test.csv > out.csv`
real time: 0.76 sec (time to read/parse all CSV records and print out new file)
- `csv-cut "1,5,6,10" test.csv > out.csv`
real time: 0.81 sec
- `csv-grep 6 "PROBLEM" test.csv > out.csv`
real time: 0.69 sec (regex matching, matches 59,936 records)
- `csv-grep -F 6 "PROBLEM" test.csv > out.csv`
real time: 0.63 sec (substring matching, matches 59,936 records)
- `csv-grep 6 "Disease" test.csv > out.csv`
real time: 0.79 sec (regex matching, matches 714,964 records)
- `csv-grep -F 6 "Disease" test.csv > out.csv`
real time: 0.73 sec (substring matching, matches 714,964 records)
- `csv-grep-cond 6 'match($0,"PROBLEM")' test.csv > out.csv` real time: 5.1 sec (equivalent of first `csv-grep` example)
- `csv-paste test.csv test.csv > out.csv`
real time: 1.47 sec
- `csv-process-field -e '{} ' 7 test.csv > out.csv`
real time: 5.2 sec (baseline minimum, since no actions)
- `csv-process-field -e '{split($0,arr,".");$0=arr[1]}' 7 test.csv > out.csv`
real time: 5.8 sec
- `csv-sort 1 test.csv > out.csv`
real time: 2.2 sec
- `csv-sort -n 1 test.csv > out.csv`
real time: 2.3 sec

- `csv-sort -D"%Y-%m-%d %T" 4 test.csv > out.csv`
real time: 3.0 sec
- `csv-sort-mult -k1 test.csv > out.csv`
real time: 3.3 sec
- `csv-sort-mult -k7 test.csv > out.csv`
real time: 3.7 sec
- `csv-sort-mult -k1 -k7 test.csv > out.csv`
real time: 8.8 sec

All CSV utils programs rely on library functions, which ultimately rely on base functions that read and parse CSV file syntax. The speed of these functions thus determines ultimate performance. Tests were run by directly calling the base read/parse functions, and without doing any output. The following average minimum runtimes were found:

- base “skip” function (merely reads/parses, does not save records):
real time: 0.522 sec
- base “read” function (reads/parses and saves records):
real time: 0.572 sec
- base “parse” function (reads/parses and provides fields):
real time: 0.617 sec