








[Discover Packages](#) > [Standard library](#) > [encoding](#) > [csv](#) **CSV**

package

standard library

Version: [go1.20.1](#) **Latest** | Published: Feb 14, 2023 | License: [BSD-3-Clause](#) | Imports: 8 |

Imported by: 17,050

**Details** Valid [go.mod](#) file   Redistributable license   Tagged version  Stable version [Learn more](#)**Repository**[cs.opensource.google/go/go](#)**Links** [Report a Vulnerability](#) Documentation <> **Documentation****Overview**

Package `csv` reads and writes comma-separated values (CSV) files. There are many kinds of CSV files; this package supports the format described in [RFC 4180](#).

A `csv` file contains zero or more records of one or more fields per record. Each record is separated by the newline character. The final record may optionally be followed by a newline character.

```
field1,field2,field3
```

White space is considered part of a field.

Carriage returns before newline characters are silently removed.

Blank lines are ignored. A line with only whitespace characters (excluding the ending newline character) is not considered a blank line.

Fields which start and stop with the quote character `"` are called quoted-fields. The beginning and ending quote are not part of the field.

The source:

```
normal string, "quoted-field"
```

results in the fields

```
{`normal string`, `quoted-field`}
```

Within a quoted-field a quote character followed by a second quote character is considered a single quote.

```
"the ""word"" is true","a ""quoted-field"""
```

results in

```
{`the "word" is true`, `a "quoted-field"`}
```

Newlines and commas may be included in a quoted-field

```
"Multi-line  
field","comma is ,"
```

results in

```
{`Multi-line  
field`, `comma is ,`}
```

## Index

### [Variables](#)

#### [type ParseError](#)

```
func (e *ParseError) Error() string  
func (e *ParseError) Unwrap() error
```

#### [type Reader](#)

```
func NewReader(r io.Reader) *Reader  
func (r *Reader) FieldPos(field int) (line, column int)  
func (r *Reader) InputOffset() int64  
func (r *Reader) Read() (record []string, err error)  
func (r *Reader) ReadAll() (records [][]string, err error)
```

#### [type Writer](#)

```
func NewWriter(w io.Writer) *Writer  
func (w *Writer) Error() error  
func (w *Writer) Flush()  
func (w *Writer) Write(record []string) error  
func (w *Writer) WriteAll(records [][]string) error
```

## Examples

### [Reader](#)

#### [Reader \(Options\)](#)

#### [Reader.ReadAll](#)

### [Writer](#)

#### [Writer.WriteAll](#)

## Constants

This section is empty.

## Variables

[View Source](#)

```
var (  
    ErrBareQuote   = errors.New("bare \" in non-quoted-field")  
    ErrQuote       = errors.New("extraneous or missing \" in quoted-field")  
    ErrFieldCount  = errors.New("wrong number of fields")  
  
    // Deprecated: ErrTrailingComma is no longer used.  
    ErrTrailingComma = errors.New("extra delimiter at end of line")  
)
```

These are the errors that can be returned in `ParseError.Err`.

## Functions

This section is empty.

## Types

### type `ParseError`

```
type ParseError struct {  
    StartLine int // Line where the record starts  
    Line      int // Line where the error occurred  
    Column    int // Column (1-based byte index) where the error occurred  
    Err       error // The actual error  
}
```

A `ParseError` is returned for parsing errors. Line numbers are 1-indexed and columns are 0-indexed.

### func (\*`ParseError`) `Error`

```
func (e *ParseError) Error() string
```

### func (\*`ParseError`) `Unwrap`

added in go1.13

```
func (e *ParseError) Unwrap() error
```

### type `Reader`

```
type Reader struct {  
    // Comma is the field delimiter.  
    // It is set to comma (',') by NewReader.  
    // Comma must be a valid rune and must not be \r, \n,  
    // or the Unicode replacement character (0xFFFD).  
}
```

Comma [rune](#)

```
// Comment, if not 0, is the comment character. Lines beginning with the
// Comment character without preceding whitespace are ignored.
// With leading whitespace the Comment character becomes part of the
// field, even if TrimLeadingSpace is true.
// Comment must be a valid rune and must not be \r, \n,
// or the Unicode replacement character (0xFFFD).
// It must also not be equal to Comma.
```

Comment [rune](#)

```
// FieldsPerRecord is the number of expected fields per record.
// If FieldsPerRecord is positive, Read requires each record to
// have the given number of fields. If FieldsPerRecord is 0, Read sets it to
// the number of fields in the first record, so that future records must
// have the same field count. If FieldsPerRecord is negative, no check is
// made and records may have a variable number of fields.
```

FieldsPerRecord [int](#)

```
// If LazyQuotes is true, a quote may appear in an unquoted field and a
// non-doubled quote may appear in a quoted field.
```

LazyQuotes [bool](#)

```
// If TrimLeadingSpace is true, leading white space in a field is ignored.
// This is done even if the field delimiter, Comma, is white space.
```

TrimLeadingSpace [bool](#)

```
// ReuseRecord controls whether calls to Read may return a slice sharing
// the backing array of the previous call's returned slice for performance.
// By default, each call to Read returns newly allocated memory owned by the caller
```

ReuseRecord [bool](#)

```
// Deprecated: TrailingComma is no longer used.
```

TrailingComma [bool](#)

```
// contains filtered or unexported fields
```

```
}
```

A Reader reads records from a CSV-encoded file.

As returned by `NewReader`, a Reader expects input conforming to [RFC 4180](#). The exported fields can be changed to customize the details before the first call to `Read` or `ReadAll`.

The Reader converts all `\r\n` sequences in its input to plain `\n`, including in multiline field values, so that the returned data does not depend on which line-ending convention an input file uses.

► [Example](#)

► [Example \(Options\)](#)

**func** [NewReader](#)

```
func NewReader(r io.Reader) *Reader
```

NewReader returns a new Reader that reads from r.

## func (\*Reader) FieldPos

added in go1.17

```
func (r *Reader) FieldPos(field int) (line, column int)
```

FieldPos returns the line and column corresponding to the start of the field with the given index in the slice most recently returned by Read. Numbering of lines and columns starts at 1; columns are counted in bytes, not runes.

If this is called with an out-of-bounds index, it panics.

## func (\*Reader) InputOffset

added in go1.19

```
func (r *Reader) InputOffset() int64
```

InputOffset returns the input stream byte offset of the current reader position. The offset gives the location of the end of the most recently read row and the beginning of the next row.

## func (\*Reader) Read

```
func (r *Reader) Read() (record []string, err error)
```

Read reads one record (a slice of fields) from r. If the record has an unexpected number of fields, Read returns the record along with the error ErrFieldCount. Except for that case, Read always returns either a non-nil record or a non-nil error, but not both. If there is no data left to be read, Read returns nil, io.EOF. If ReuseRecord is true, the returned slice may be shared between multiple calls to Read.

## func (\*Reader) ReadAll

```
func (r *Reader) ReadAll() (records [][]string, err error)
```

ReadAll reads all the remaining records from r. Each record is a slice of fields. A successful call returns err == nil, not err == io.EOF. Because ReadAll is defined to read until EOF, it does not treat end of file as an error to be reported.

► [Example](#)

## type Writer

```
type Writer struct {
    Comma    rune // Field delimiter (set to ',' by NewWriter)
    UseCRLF  bool // True to use \r\n as the line terminator
    // contains filtered or unexported fields
}
```

A `Writer` writes records using CSV encoding.

As returned by `NewWriter`, a `Writer` writes records terminated by a newline and uses `'` as the field delimiter. The exported fields can be changed to customize the details before the first call to `Write` or `WriteAll`.

Comma is the field delimiter.

If `UseCRLF` is true, the `Writer` ends each output line with `\r\n` instead of `\n`.

The writes of individual records are buffered. After all data has been written, the client should call the `Flush` method to guarantee all data has been forwarded to the underlying `io.Writer`. Any errors that occurred should be checked by calling the `Error` method.

► [Example](#)

### **func** `NewWriter`

```
func NewWriter(w io.Writer) *Writer
```

`NewWriter` returns a new `Writer` that writes to `w`.

### **func** (`*Writer`) `Error`

added in go1.1

```
func (w *Writer) Error() error
```

`Error` reports any error that has occurred during a previous `Write` or `Flush`.

### **func** (`*Writer`) `Flush`

```
func (w *Writer) Flush()
```

`Flush` writes any buffered data to the underlying `io.Writer`. To check if an error occurred during the `Flush`, call `Error`.

### **func** (`*Writer`) `Write`

```
func (w *Writer) Write(record []string) error
```

`Write` writes a single CSV record to `w` along with any necessary quoting. A record is a slice of strings with each string being one field. Writes are buffered, so `Flush` must eventually be called to ensure that the record is written to the underlying `io.Writer`.

### **func** (`*Writer`) `WriteAll`

```
func (w *Writer) WriteAll(records [][]string) error
```

WriteAll writes multiple CSV records to w using Write and then calls Flush, returning any error from the Flush.

► [Example](#)

## Source Files

[View all](#) 

[reader.go](#)

[writer.go](#)

### Why Go

[Use Cases](#)

[Case Studies](#)

### Get Started

[Playground](#)

[Tour](#)

[Stack Overflow](#)

[Help](#)

### Packages

[Standard Library](#)

[About Go Packages](#)

### About

[Download](#)

[Blog](#)

[Issue Tracker](#)

[Release Notes](#)

[Brand Guidelines](#)

[Code of Conduct](#)

### Connect

[Twitter](#)

[GitHub](#)

[Slack](#)

[r/golang](#)

[Meetup](#)

[Golang Weekly](#)

---

[Copyright](#)

[Terms of Service](#)

[Privacy Policy](#)

[Report an Issue](#)



Google