- DISCLAIMER
- NAME
- SYNOPSIS
- DESCRIPTION
  - Embedded newlines
  - Binary data
- SPECIFICATION
- METHODS
  - version
  - new
  - print
  - say
  - combine
  - string
  - getline
  - getline_all =head2 getline-all
  - getline_hr =head2 getline-hr
  - getline_hr_all =head2 getline-hr-all
  - parse
  - fragment
  - colrange
  - rowrange
  - column_names =head2 column-names
  - header
    - return value
    - Options
  - bind_columns =head2 bind-columns
  - eof
  - types
  - row
  - fields
  - strings
  - meta_info
  - meta-info
  - is_quoted
  - is-quoted
  - is_binary
  - is-binary
  - is_missing
  - is-missing
  - status
  - error_input
  - error-input
- DIAGNOSTICS
  - CSV::Diag
  - error_diag
  - error-diag
  - record_number
  - set_diag
- CSV::Field

        - new
        - Bool
        - text =head2 Str
        - Buf
        - Numeric
        - gist
        - add
        - set_quoted
        - is_quoted
        - undefined
        - is_binary
        - is_utf8
        - is_missing
- CSV::Row
    - methods
- FUNCTIONS
    - csv
        - in
        - out
        - encoding
        - detect-bom
        - headers
        - munge-column-names
        - key
        - fragment
        - sep-set
        - set_column_names
    - Callbacks
        - Callbacks for csv
- EXAMPLES
    - Reading a CSV file line by line:
        - Reading only a single column
    - Parsing CSV strings:
    - Printing CSV data
        - The fast way: using "print"
        - The slow way: using "combine" and "string"
    - Rewriting CSV
    - Dumping database tables to CSV
    - The examples folder
- CAVEATS
    - Microsoft Excel
- TODO / WIP / NYI
- DIAGNOSTICS
- SEE ALSO
- AUTHOR
- COPYRIGHT AND LICENSE

DISCLAIMER

Note that updating these docs is an ongoing process and some perl5 idioms might not have been translated yet

into correct raku idiom. My bad. Sorry. (Feedback welcome)

NAME

Text::CSV - comma-separated values manipulation routines

SYNOPSIS

```
use Text::CSV;

# Read whole file in memory
my @aoa = csv(in => "data.csv");  # as array of arrays
my @aoh = csv(in => "data.csv",
              headers => "auto"); # as array of hashes

# Write array of arrays as csv file
csv(in => @aoa, out => "file.csv", sep => ";");

my @rows;
# Read/parse CSV
my $csv = Text::CSV.new;
my $fh  = open "test.csv", :r, :!chomp;
while (my @row = $csv.getline($fh)) {
    @row[2] ~~ m/pattern/ or next; # 3rd field should match
    @rows.push: @row;
    }
$fh.close;

# and write as CSV
$fh = open "new.csv", :w;
$csv.say($fh, $_) for @rows;
$fh.close;
```

DESCRIPTION

Text::CSV provides facilities for the composition and decomposition of comma-separated values. An instance of the Text::CSV class will combine fields into a `csv` string and parse a `csv` string into fields.

The module accepts either strings or files as input and support the use of user-specified characters (or sequences thereof) for delimiters, separators, and escapes.

In all following documentation, `WIP` stands for "Work In Progress" and `NYI` for "Not Yet Implemented". The goal is to get rid of all of those.

Embedded newlines

Important Note: The default behavior is to accept only UTF-8 characters.

But you still have the problem that you have to pass a correct line to the "parse" method, which is more complicated from the usual point of usage:

```
my $csv = Text::CSV.new;
for lines() : eager {          #  WRONG!
    $csv.parse($_);
    my @fields = $csv.fields;
    }
```

this will break for several reasons. The default open mode is to `chomp` lines, which will also remove the newline sequence if that sequence is not (part of) the newline at all. As the `for` might read broken lines: it does not care about the quoting. If you need to support embedded newlines, the way to go is to **not** pass `eol` in the parser (it accepts `\n`, `\r`, **and** `\r\n` by default) and then

```
my $csv = Text::CSV.new;
my $io = open $file, :r, :!chomp;
while (my $row = $csv.getline($io)) {
    my @fields = @$row;
    }
```

Binary data

For now, Text::CSV only accepts Unicode. Binary data is planned.

SPECIFICATION

While no formal specification for CSV exists, RFC 4180 (*1*) describes the common format and establishes `text/csv` as the MIME type registered with the IANA. RFC 7111 (*2*) adds fragments to CSV.

Many informal documents exist that describe the CSV format. "How To: The Comma Separated Value (CSV) File Format" (*3*) provides an overview of the CSV format in the most widely used applications and explains how it can best be used and supported.

```
1) http://tools.ietf.org/html/rfc4180
2) http://tools.ietf.org/html/rfc7111
3) http://www.creativyst.com/Doc/Articles/CSV/CSV01.htm
```

The basic rules are as follows:

CSV is a delimited data format that has fields/columns separated by the comma character and records/rows separated by newlines. Fields that contain a special character (comma, newline, or double quote), must be enclosed in double quotes. However, if a line contains a single entry that is the empty string, it may be enclosed in double quotes. If a field's value contains a double quote character it is escaped by placing another double quote character next to it. The CSV file format does not require a specific character encoding, byte order, or line terminator format.

- Each record is a single line ended by a line feed (ASCII/`LF=0x0A`) or a carriage return and line feed pair (ASCII/`CRLF=0x0D 0x0A`), however, line-breaks may be embedded.
- Fields are separated by commas.
- Allowable characters within a CSV field include `0x09` (`TAB`) and the inclusive range of `0x20` (space) through `0x7E` (tilde). In binary mode all characters are accepted, at least in quoted fields.
- A field within CSV must be surrounded by double-quotes to contain a separator character (comma).

Though this is the most clear and restrictive definition, Text::CSV is way more liberal than this, and allows extension:

- Line termination by a single carriage return is accepted by default
- The separation, escape, and escape can be any valid Unicode sequence.
- A field in CSV must be surrounded by double-quotes to make an embedded double-quote, represented by a pair of consecutive double-quotes, valid. You may additionally use the sequence `"0` for representation of a NULL byte. Using `0x00` is just as valid.
- Several violations of the above specification may be lifted by passing some options as attributes to the object constructor.

METHODS

version

Returns the current module version.

new

Returns a new instance of class Text::CSV. The attributes are described by the optional named parameters

```
my $csv = Text::CSV.new(attributes ...);
```

The following attributes are available:

eol

```
my $csv = Text::CSV.new(eol => "\r\n");
          $csv.eol(Str);
my $eol = $csv.eol;
```

The end-of-line string to add to rows for "print" or the record separator for "getline".

When not set in a **parser** instance, the default behavior is to accept `\n`, `\r`, and `\r\n`, so it is probably safer to not specify `eol` at all. Passing `Str` or the empty string behave the same.

As raku interprets `\r\n` as a single grapheme in input, it is dissuaded to use `\r\n` as `eol` when parsing. Please choose `Str` instead.

When not passed in a **generating** instance, records are not terminated at all, so it is probably wise to pass something you expect. A safe choice for `eol` on output is either `\n` or `\r\n`.

Common values for `eol` are `"\012"` (`\n` or Line Feed), `"\015\012"` (`\r\n` or Carriage Return, Line Feed), and `"\015"` (`\r` or Carriage Return).

sep
sep_char
sep-char
separator

```
my $csv = Text::CSV.new(sep => ";");
          $csv.sep("\x[ff0c]"); # FULLWIDTH COMMA
my $sep = $csv.sep;
```

The sequence used to separate fields, by default a comma: (`,`). This sequence is required and cannot be disabled.

The separation sequence can not be equal to the quote sequence, the escape sequence or the newline sequence.

See also "CAVEATS"

quote
quote_char
quote-char

```
my $csv = Text::CSV.new(quote => "'");
          $csv.quote("\x[ff02]); # FULLWIDTH QUOTATION MARK
          $csv.quote(Str);
my $quo = $csv.quote;
```

The sequence to quote fields containing blanks or binary data, by default the double quote character (`"`). A value of `Str` disables quotation (for simple cases only).

The quotation sequence can not be equal to the separation sequence or the newline sequence.

See also "CAVEATS"

escape
escape_char
escape-char

```
my $csv = Text::CSV.new(escape => "\\");
          $csv.escape("\x[241b]");  # SYMBOL FOR ESCAPE
          $csv.escape(Str);
my $esc = $csv.escape;
```

The sequence to escape certain characters inside quoted fields.

The `escape` defaults to being the double-quote mark (`"`). In other words the same as the default sequence for `quote`. This means that doubling the quote mark in a field escapes it:

```
"foo","bar","Escape ""quote mark"" with two ""quote marks""","baz"
```

If you change `quote` without changing `escape`, `escape` will still be the double-quote (`"`). If instead you want to escape `quote` by doubling it you will need to also change `escape` to be the same as what you have changed `quote` to.

The escape sequence can not be equal to the separation sequence.

binary

WIP: Default is True. Non-UTF-8 real binary (Blob) does not yet parse. Opening the resource with encoding utf8-c8 is most likely the way to go.

```
my $csv = Text::CSV.new(:binary);
          $csv.binary(False);
my $bin = $csv.binary;
```

If this attribute is `True`, you may use binary data in quoted fields, including line feeds, carriage returns and `NULL` bytes. (The latter could be escaped as `"0`.) By default this feature is on.

Note that valid Unicode (UTF-8) is not considered binary.

strict

```
my $csv = Text::CSV.new(:strict);
          $csv.strict(False);
my $flg = $csv.strict;
```

If set to True, any row that parses to a different number of columns than the previous row will cause the parser to throw error 2014.

formula-handling
formula_handling
formula

```
my $csv = Text::CSV.new(formula => "none");
        $csv.formula("none");
my $f = $csv.formula;
```

This defines the behavior of fields containing *formulas*. As formulas are considered dangerous in spreadsheets, this attribute can define an optional action to be taken if a field starts with an equal sign (`=`).

For purpose of code-readability, this can also be written as

```
my $csv = Text::CSV_XS.new(formula-handling => "none"});
        $csv.formula-handling("none");
my $f = $csv.formula-handling;
```

or

```
my $csv = Text::CSV_XS.new(formula_handling => "none"});
        $csv.formula_handling("none");
my $f = $csv.formula_handling;
```

Possible values for this attribute are

none

Take no specific action. This is the default.

```
$csv.formula("none");
```

die

Cause the process to `die` whenever a leading `=` is encountered.

```
$csv.formula("die");
```

croak

Cause the process to `die` whenever a leading `=` is encountered.

```
$csv.formula("croak");
```

This option just exists for perl5 compatibility.

diag

Report position and content of the field whenever a leading `=` is found. The value of the field is unchanged.

```
$csv.formula("diag");
```

empty

Replace the content of fields that start with a `=` with the empty string.

```
$csv.formula("empty");
$csv.formula("");
```

undef

Replace the content of fields that start with a `=` with `Str`.

```
$csv.formula("undef");
$csv.formula(Str);
```

All other values will throw an exception with error code 1500.

auto_diag
auto-diag

```
my $csv = Text::CSV.new(auto_diag => False);
          $csv.auto_diag(True);
my $a-d = $csv.auto_diag;
```

Set this attribute to a number between `1` and `9` causes "error_diag" to be automatically called in void context upon errors. The value `True` evaluates to `1`.

In case of error `2012 - EOF`, this call will be void.

If `auto_diag` is set to a numeric value greater than `1`, it will `die` on errors instead of `warn`.

diag_verbose
diag-verbose

```
my $csv = Text::CSV.new(diag_verbose => 1);
          $csv.diag_verbose(2);
my $d-v = $csv.diag_verbose;
```

Set the verbosity of the output triggered by `auto_diag`.

WIP: Does not add any information yet.

blank_is_undef
blank-is-undef

```
my $csv = Text::CSV.new(:blank_is_undef);
          $csv.blank_is_undef(False);
my $biu = $csv.blank_is_undef;
```

Under normal circumstances, `CSV` data makes no distinction between quoted- and unquoted empty fields. These both end up in an empty string field once read, thus

```
1,"",," ",2
```

is read as

```
[ "1", "", "", " ", "2" ]
```

When *writing* `CSV` files with `always_quote` set, the unquoted *empty* field is the result of an undefined value. To enable this distinction when *reading* `CSV` data, the `blank_is_undef` attribute will cause unquoted empty fields to be set to `Str`, causing the above to be parsed as

```
[ "1", "", Str, " ", "2" ]
```

empty_is_undef
empty-is-undef

```
my $csv = Text::CSV.new(:empty_is_undef);
          $csv.empty_is_undef(False);
my $eiu = $csv.empty_is_undef;
```

Going one step further than `blank_is_undef`, this attribute causes all empty fields to return as `Str`, so

```
1,"",," ",2
```

is read as

```
[ 1, Str, Str, " ", 2 ]
```

Note that this effects only fields that are originally empty, not fields that are empty after stripping allowed whitespace. YMMV.

allow_whitespace
allow-whitespace

```
my $csv = Text::CSV.new(:allow_whitespace);
          $csv.allow_whitespace(False);
my $a-w = $csv.allow_whitespace;
```

When this option is set to `True`, the whitespace (`TAB`'s and `SPACE`'s) surrounding the separation sequence is removed when parsing. If either `TAB` or `SPACE` is one of the three major sequences `sep`, `quote`, or `escape` it will not be considered whitespace.

Now lines like:

```
1 , "foo" , bar , 3 , zapp
```

are parsed as valid csv, even though it violates the csv specs.

Note that **all** whitespace is stripped from both start and end of each field. That would make it *more* than a *feature* to enable parsing bad csv lines, as

```
1,   2.0,  3,   ape  , monkey
```

will now be parsed as

```
[ "1", "2.0", "3", "ape", "monkey" ]
```

even if the original line was perfectly acceptable csv.

allow_loose_quotes
allow-loose-quotes

```
my $csv = Text::CSV.new(:allow_loose_quotes);
          $csv.allow_loose_quotes(False);
my $alq = $csv.allow_loose_quotes;
```

By default, parsing unquoted fields containing `quote`'s like

```
1,foo "bar" baz,42
```

would result in parse error 2034. Though it is still bad practice to allow this format, we cannot help the fact that some vendors make their applications spit out lines styled this way.

If there is **really** bad csv data, like

```
1,"foo "bar" baz",42
```

or

```
1,""foo bar baz"",42
```

there is a way to get this data-line parsed and leave the quotes inside the quoted field as-is. This can be achieved by setting `allow_loose_quotes` **AND** making sure that the `escape` is *not* equal to `quote`.

allow_loose_escapes
allow-loose-escapes

```
my $csv = Text::CSV.new(:allow_loose_escapes);
          $csv.allow_loose_escapes(False);
my $ale = $csv.allow_loose_escapes;
```

Parsing fields that have `escape` sequences that escape characters that do not need to be escaped, like:

```
my $csv = Text::CSV.new(escape_char => "\\");
$csv.parse(q{1,"my bar\'s",baz,42});
```

would result in parse returning `False` with reason 2025. Though it is bad practice to allow this format, this attribute enables you to treat all escape sequences equal.

allow_unquoted_escape
allow-unquoted-escape

```
my $csv = Text::CSV.new(:allow_unquoted_escape);
          $csv.allow_unquoted_escape(False);
my $aue = $csv.allow_unquoted_escape;
```

A backward compatibility issue where `escape` differs from `quote` prevents `escape` to be in the first position of a field. If `quote` is equal to the default `"` and `escape` is set to `\`, this would be illegal:

```
1,\0,2
```

Setting this attribute to `True` might help to overcome issues with backward compatibility and allow this style.

always_quote
always-quote

```
my $csv = Text::CSV.new(:always_quote);
       $csv.always_quote(False);
my $f = $csv.always_quote;
```

By default the generated fields are quoted only if they *need* to be. For example, if they contain the separator sequence. If you set this attribute to `1` then *all* defined fields will be quoted. (undefined (`Str`) fields are not quoted, see "blank_is_undef"). This makes it quite often easier to handle exported data in external applications. (Poor creatures who are better to use Text::CSV. :)

quote_empty
quote-empty

```
my $csv = Text::CSV.new(:quote_empty);
          $csv.quote_empty(False);
my $q-s = $csv.quote_empty;
```

By default the generated fields are quoted only if they *need* to be. An empty defined field does not need quotation. If you set this attribute to `True` then empty defined fields will be quoted. See also `always_quote`.

quote_space
quote-space

```
my $csv = Text::CSV.new(:quote_space);
          $csv.quote_space(False);
my $q-s = $csv.quote_space;
```

By default, a space in a field would trigger quotation. As no rule exists this to be forced in `CSV`, nor any for the opposite, the default is `True` for safety. You can exclude the space from this trigger by setting this attribute to `False`.

escape_null
quote-null

```
my $csv = Text::CSV.new(:escape_null);
          $csv.escape_null(False);
my $q-n = $csv.escape_null;
```

By default, a `NULL` byte in a field would be escaped. This option enables you to treat the `NULL` byte as a simple binary character in binary mode (the `binary => True` is set). The default is `True`. You can prevent `NULL` escapes by setting this attribute to `False`.

quote_binary
quote-binary

```
my $csv = Text::CSV.new(:quote_binary);
          $csv.quote_binary(False);
my $q-b = $csv.quote_binary;
```

By default, all "unsafe" bytes inside a string cause the combined field to be quoted. By setting this attribute to `False`, you can disable that trigger for bytes `>= 0x7F`. (WIP)

keep_meta
keep-meta

```
my $csv = Text::CSV.new(:keep_meta);
          $csv.keep_meta(False);
my $k-m = $csv.keep_meta_info;
```

By default, the parsing of input records is as simple and fast as possible. However, some parsing information - like quotation of the original field - is lost in that process. Setting this flag to true enables retrieving that information after parsing with the methods "meta_info", "is_quoted", and "is_binary" described below. Default is `False` for ease of use.

If `keep-meta` is set to `True`, the returned fields are not of type `Str` but of type `CSV::Field`.

types

NYI

A set of column types; the attribute is immediately passed to the "types" method.

callbacks

See the "Callbacks" section below.

To sum it up,

```
$csv = Text::CSV.new;
```

is equivalent to

```
$csv = Text::CSV.new(
    eol                  => Nil, # \r, \n, or \r\n
    sep                  => ',',
    quote                => '"',
    escape               => '"',
    binary               => True,
    auto-diag            => False,
    diag-verbose         => 0,
    blank-is-undef       => False,
    empty-is-undef       => False,
    allow-whitespace     => False,
    allow-loose-quotes   => False,
    allow-loose-escapes  => False,
    allow-unquoted-escape => False,
    always-quote         => False,
    quote-space          => True,
    escape-null          => True,
    quote-binary         => True,
    keep-meta            => False,
    strict               => False,
    formula              => "none",
    undef-str            => Str,
    types                => Nil,
    callbacks            => Nil,
    });
```

For all of the above mentioned flags, an accessor method is available where you can inquire the current value, or change the value

```
my $quote = $csv.quote;
$csv.binary(True);
```

It is not wise to change these settings halfway through writing `csv` data to a stream. If however you want to create a new stream using the available `csv` object, there is no harm in changing them.

If the "new" constructor call fails, an exception of type `csv::Diac` is thrown with the reason like the "error_diag" method would return:

```
my $e;
{   $csv = Text::CSV.new(ecs_char => ":") or
    CATCH { default { $e = $_; }}
    }
$e and $e.message.say;
```

The message will be a string like

```
"INI - Unknown attribute 'ecs_char'"
```

print

```
$status = $csv.print($io,  $fld, ... );
$status = $csv.print($io, ($fld, ...));
$status = $csv.print($io, [$fld, ...]);
$status = $csv.print($io,  @fld      );

$csv.column_names(%fld.keys); # or use a subset
$status = $csv.print($io,  %fld      );
```

Similar to "combine" + "string" + "print", but much more efficient. It takes an IO object and any number of arguments interpreted as fields. The resulting string is immediately written to the `$io` stream.

NYI: no fields in combination with "bind_columns", like

```
$csv.bind_columns(\($foo, $bar));
$status = $csv.print($fh);
```

A benchmark showed this order of preference, but the difference is within noise range:

```
my @data = ^20;
$csv.print($io,   @data  );   # 2.6 sec
$csv.print($io, [ @data ]);   # 2.7 sec
$csv.print($io,    ^20   );   # 2.7 sec
$csv.print($io,  \@data  );   # 2.8 sec
```

say

Is the same a s"print" where `eol` defaults to `$*OUT.nl`.

```
$status = $csv.say($io,  $fld, ... );
$status = $csv.say($io, ($fld, ...));
$status = $csv.say($io, [$fld, ...]);
$status = $csv.say($io,  @fld      );

$csv.column_names(%fld.keys); # or use a subset
$status = $csv.say($io,  %fld      );
```

combine

```
$status = $csv.combine(@fields);
$status = $csv.combine($fld, ...);
$status = $csv.combine(\@fields);
```

This method constructs a `csv` row from `@fields`, returning success or failure. Failure can result from lack of arguments or an argument that contains invalid data. Upon success, "string" can be called to retrieve the resultant `csv` string. Upon failure, the value returned by "string" is undefined and "error_input" could be called to retrieve the invalid argument. (WIP)

string

```
$line = $csv.string;
```

This method returns the input to "parse" or the resultant `csv` string of "combine", whichever was called more recently. If `eol` is defined, it is added to the string.

getline

```
@row = $csv.getline($io);
@row = $csv.getline($io,  :meta);
@row = $csv.getline($str);
@row = $csv.getline($str, :meta);
```

This is the counterpart to "print", as "parse" is the counterpart to "combine": it parses a row from the `$io` handle or `$str` using the "getline" method associated with `$io` (or the internal temporary IO handle used to read from the string as if it were an IO handle) and parses this row into an array. This array is returned by the function or `Array` for failure. When `$io` does not support `getline`, you are likely to hit errors.

NYI: When fields are bound with "bind_columns" the return value is a reference to an empty list.

getline_all =head2 getline-all

```
@rows = $csv.getline_all($io);
@rows = $csv.getline_all($io,                  :meta);
@rows = $csv.getline_all($io, $offset);
@rows = $csv.getline_all($io, $offset,         :meta);
@rows = $csv.getline_all($io, $offset, $length);
@rows = $csv.getline_all($io, $offset, $length, :meta);
```

This will return a list of getline($io) results. If `$offset` is negative, as with `splice`, only the last `abs($offset)` records of `$io` are taken into consideration.

Given a CSV file with 10 lines:

```
lines call
----- ----------------------------------------------------------
0..9  $csv.getline_all($io)         # all
0..9  $csv.getline_all($io,  0)     # all
8..9  $csv.getline_all($io,  8)     # start at 8
-     $csv.getline_all($io,  0,  0) # start at 0 first 0 rows
0..4  $csv.getline_all($io,  0,  5) # start at 0 first 5 rows
4..5  $csv.getline_all($io,  4,  2) # start at 4 first 2 rows
8..9  $csv.getline_all($io, -2)     # last 2 rows
6..7  $csv.getline_all($io, -4,  2) # first 2 of last  4 rows
```

getline_hr =head2 getline-hr

The "getline_hr" and "column_names" methods work together to allow you to have rows returned as hashes instead of arrays. You must invoke "column_names" first to declare your column names.

```
$csv.column_names(< code name price description >);
%hr = $csv.getline_hr($str, :meta);
%hr = $csv.getline_hr($io);
say "Price for %hr<name> is %hr<price> \c[EURO SIGN]";
```

"getline_hr" will fail if invoked before "column_names".

getline_hr_all =head2 getline-hr-all

```
@rows = $csv.getline_hr_all($io);
@rows = $csv.getline_hr_all($io,                  :meta);
@rows = $csv.getline_hr_all($io, $offset);
@rows = $csv.getline_hr_all($io, $offset,         :meta);
@rows = $csv.getline_hr_all($io, $offset, $length);
@rows = $csv.getline_hr_all($io, $offset, $length, :meta);
```

This will return a list of getline_hr($io) results.

parse

```
$status = $csv.parse($line);
```

This method decomposes a csv string into fields, returning success or failure. Failure can result from a lack of argument or improper format in the given csv string. Upon success, invoke "fields" or "strings" to get the decomposed fields. Upon failure these methods shall not be trusted to return reliable data.

NYI: You may use the "types" method for setting column types. See "types"' description below.

fragment

This function implements RFC7111 (URI Fragment Identifiers for the text/csv Media Type).

```
my @rows = $csv.fragment($io, $spec);
```

In specifications, `*` is used to specify the *last* item, a dash (`-`) to indicate a range. All indices are `1`-based: the first row or column has index `1`. Selections can be combined with the semi-colon (`;`).

When using this method in combination with "column_names", the returned array will be a list of hashes instead of an array of arrays. A disjointed cell-based combined selection might return rows with different number of columns making the use of hashes unpredictable.

```
$csv.column_names(< Name Age >);
my @rows = $csv.fragment($io, "row=3;8");
```

Note that for `col=".."`, the column names are the names for *before* the selection is taken to make it more consistent with reading possible headers from the first line of the CSV datastream.

```
$csv,column_names(< foo bar >);  # WRONG
$csv.fragment($io, "col=3");
```

would set the column names for the first two columns that are then skipped in the fragment. To skip the unwanted columns, use placeholders.

```
$csv.column_names(< x x Name >);
$csv.fragment($io, "col=3");
```

WIP: If the "after_parse" callback is active, it is also called on every line parsed and skipped before the fragment.

row

```
row=4
row=5-7
row=6-*
row=1-2;4;6-*
```

col

```
col=2
col=1-3
col=4-*
col=1-2;4;7-*
```

cell

In cell-based selection, the comma (`,`) is used to pair row and column

```
cell=4,1
```

The range operator (`-`) using `cell`s can be used to define top-left and bottom-right `cell` location

```
cell=3,1-4,6
```

The `*` is only allowed in the second part of a pair

```
cell=3,2-*,2    # row 3 till end, only column 2
cell=3,2-3,*    # column 2 till end, only row 3
cell=3,2-*,*    # strip row 1 and 2, and column 1
```

Cells and cell ranges may be combined with `;`, possibly resulting in rows with different number of columns

```
cell=1,1-2,2;3,3-4,4;1,4;4,1
```

Disjointed selections will only return selected cells. The cells that are not specified will not be included in the returned set, not even as `Str`. As an example given a `CSV` like

```
11,12,13,...19
21,22,...28,29
:            :
91,...97,98,99
```

with `cell=1,1-2,2;3,3-4,4;1,4;4,1` will return:

```
11,12,14
21,22
33,34
41,43,44
```

Overlapping cell-specs will return those cells only once, So `cell=1,1-3,3;2,2-4,4;2,3;4,2` will return:

```
11,12,13
21,22,23,24
31,32,33,34
42,43,44
```

RFC7111 does **not** allow different types of specs to be combined (either `row` *or* `col` *or* `cell`). Passing an invalid fragment specification will croak and set error 2013.

Using "colrange" and "rowrange" instead of "fragment" will allow you to combine row- and column selecting as a grid.

colrange

```
my Int @range = ^5, 5..9;
$csv.colrange(@range);
$csv.colrange("0-4;6-10");
my @range = $csv.colrange;
```

Set or inspect the column ranges. When passed as an array of `Int`, the indexes are 0-based. When passed as a string, the syntax of the range is as defined by RFC7111 and thus 1-based.

rowrange

```
$csv.rowrange("1;16-*");
my @r = $csv.rowrange;
```

Set or inspect the row ranges. Only supports RFC7111 style. Indexes are 1-based.

column_names =head2 column-names

Set the "keys" that will be used in the "getline_hr" calls. If no keys (column names) are passed, it will return the current setting as a list.

```
$csv.column_names(< code description price >);
my @names = $csv.column_names;
```

"column_names" accepts a list of strings (the column names) or a single array with the names. You can pass the return value from "getline" too:

```
$csv.column_names($csv.getline($io));
```

"column_names" does **no** checking on duplicates at all, which might lead to unexpected results. As raku does not accept undefined keys in a hash, passing just types will lead to fail later on.

```
$csv.column_names(Str, "", "name"); # Will FAIL becaus of Str
$csv.column_names(< code name count name >); # will drop the second column
%hr = $csv.getline_hr($io);
```

header

Parse the CSV header and set `sep` and encoding.

```
my @hdr = $csv.header($fh).column-names;
$csv.header($fh, sep-set => [ ";", ",", "|", "\t" ]);
$csv.header($fh, munge-column-names => "fc");
```

The first argument should be a file handle.

Assuming that the file opened for parsing has a header, and the header does not contain problematic characters like embedded newlines, read the first line from the open handle then auto-detect whether the header separates the column names with a character from the allowed separator list.

If any of the allowed separators matches, and none of the *other* allowed separators match, set `sep` to that separator for the current CSV_XS instance and use it to parse the first line, map those to lowercase, and use that to set the instance "column_names":

```
my $csv = Text::CSV.new;
my $fh = open "file.csv";
$csv.header($fh);
while (my $row = $csv.getline_hr($fh)) {
    ...
    }
```

If the header is empty, contains more than one unique separator out of the allowed set, contains empty fields, or contains identical fields (after folding), it will croak with error 1010, 1011, 1012, or 1013 respectively.

If the header contains embedded newlines or is not valid CSV in any other way, this method will throw an exception.

A successful call to `header` will always set the `sep` of the `$csv` object. This behavior can not be disabled.

return value

On error this method will throw an exception.

On success, this method will return the instance.

Options

sep-set

```
$csv.header($fh, sep_set => [ ";", ",", "|", "\t" ]);
```

The list of legal separators defaults to `[ ";", "," ]` and can be changed by this option.

Multi-byte sequences are allowed, both multi-character and Unicode. See `sep`.

munge-column-names

This option offers the means to modify the column names into something that is most useful to the application. The default is to map all column names to fold case.

```
$csv.header($fh, munge-column-names => "lc");
```

The following values are available:

```
fc   - fold case
lc   - lower case
uc   - upper case
none - do not change
&cb  - supply a callback
```

```
$csv.header($fh, munge-column-names => { "column_".$col++ });
```

set-column-names

```
$csv.header($fh, :set-column-names);
```

> The default is to set the instances column names using "column_names" if the method is successful, so subsequent calls to "getline_hr" can return a hash. Disable setting the header can be forced using a false value for this option like `:!set-column-names`.

bind_columns =head2 bind-columns

NYI!

Takes a list of scalar references to be used for output with "print" or to store in the fields fetched by "getline". When you do not pass enough references to store the fetched fields in, "getline" will fail with error `3006`. If you pass more than there are fields to return, the content of the remaining references is left untouched.

```
$csv.bind_columns(\$code, \$name, \$price, \$description);
while ($csv.getline($io)) {
    print "The price of a $name is \x[20ac] $price\n";
    }
```

To reset or clear all column binding, call "bind_columns" with the single undefined argument like `Array`. This will also clear column names.

```
$csv.bind_columns(Array);
```

If no arguments are passed at all, "bind_columns" will return the list of current bindings or `Array` if no binds are active.

eof

```
$eof = $csv.eof;
```

If "parse" or "getline" was used with an IO stream, this method will return `True` if the last call hit end of file, otherwise it will return `False`. This is useful to see the difference between a failure and end of file.

If the last "parse" or "getline" finished just before end of file, the **next** "parse" or "getline" will fail and set `eof`.

That implies that if you are *not* using "auto-diag", an idiom like

```
while (my @row = $csv.getline ($fh)) {
    # ...
    }
$csv.eof or $csv.error_diag;
```

will *not* report the error. You would have to change that to

```
while (my @row = $csv.getline ($fh)) {
    # ...
    }
$csv.error_diag.error and $csv.error_diag;
```

types

NYI!

```
$csv.types(@types);
my @types = $csv.types;
```

This method is used to force a type for all fields in a column. For example, if you have an integer column, two

columns with doubles and a string column, then you might do a

```
$csv.types(Int, Num, Num, Str);
```

You can unset column types by doing a

```
$csv.types(Array);
```

row

```
CSV::Row $row = $csv.row;
```

Returns the last row parsed. See `CSV::Row`

fields

```
CSV::Field @fields = $csv.fields;
```

This method returns the input to "combine" or the resultant decomposed fields of a successful "parse" or "getline", whichever was called more recently. The fields are still of type `CSV::Field` and thus feature attributes.

strings

```
@fields = $csv.strings;
```

This method returns the input to "combine" or the resultant decomposed fields of a successful "parse" or "getline", whichever was called more recently. The fields are simplified to Str entries from `CSV::Field`, so no attributes ate available.

NYI: If types are used, the fields should comply to the types.

meta_info

meta-info

```
$csv.meta_info(True);
my $km = $csv.keep_meta;
```

This methods sets or inquires the default setting for keeping meta-info on fields. See `CSV::Field`.

is_quoted

is-quoted

```
my $quoted = $csv.is_quoted($column_idx);
```

Where `$column_idx` is the (zero-based) index of the column in the last result of "parse" or "getline", even if `meta` was false on that last invocation.

This returns `True` if the data in the indicated column was enclosed in `quote_char` quotes. This might be important for fields where content `,20070108,` is to be treated as a numeric value, and where `,"20070108",` is explicitly marked as character string data.

Also see `CSV::Field`.

is_binary

is-binary

NYI/WIP: utf8-c8

```
my $binary = $csv.is_binary($column_idx);
```

Where `$column_idx` is the (zero-based) index of the column in the last result of "parse" or "getline", even if `meta` was false on that last invocation.

Also see `CSV::Field`.

is_missing

is-missing

NYI

```
my $missing = $csv.is_missing($column_idx);
```

Where `$column_idx` is the (zero-based) index of the column in the last result of "parse" or "getline", even if `meta` was false on that last invocation.

```
while (my @row = $csv.getline_hr($fh, :meta)) {
    $csv.is_missing(0) and next; # This was an empty line
    }
```

When using "getline_hr", it is impossible to tell if the parsed fields are undefined because they where not filled in the CSV stream or because they were not read at all, as **all** the fields defined by "column_names" are set in the hash-ref. If you still need to know if all fields in each row are provided, you should enable `keep_meta` so you can check the flags.

status

```
$status = $csv.status;
```

This method returns success (or failure) of the last invoked "combine" or "parse" call.

error_input

error-input

```
$bad_argument = $csv.error_input;
```

This method returns the erroneous argument (if it exists) of "combine", "parse", or "getline", whichever was called most recent. If the last invocation was successful, `error_input` will return `Str`.

DIAGNOSTICS

Errors are transported internally using the `CSV::Diag` class. Text::CSV will return that object when it fails, so it can be caught, but on non-fatal failures, like parse returning `False`, one can use the methods to inquire the internal status.

CSV::Diag

The method is created with the widest possible use in mind, serving both the mindset of raku as well as the direct approach of the old module. It is immutable: it is created with all available error parameters known at the time of failure, and the cannot be changed afterwards.

```
my CSV::Diag $d .= new(
    error   => 0,
    message => "",
    pos     => 0,
    field   => 0,
    record  => 0,
    buffer  => Str
    );
```

If only `error` is given, the message is set accordingly if it is a known error value.

The object can be used in many contexts:

void context

```
CSV::Diag.new(error => 2034, buffer => q{ "",}, pos => 1);
```

will print

```
EIF - Loose unescaped quote : error 2034 @ record 1, field 1, position 2
 "?",
```

which is what happens when `auto_diag` is `True` and you parse illegal CSV:

```
Text::CSV.new(:auto_diag).parse(q{ "",});'
EIF - Loose unescaped quote : error 2034 @ record 1, field 1, position 2
 "?",
```

numeric context

Will return the error code

```
my Int $e;
{   fail CSV::Diag(error => 2034);
    CATCH { default { $e = +$_; }}
    }
# $e is now 2034
```

string context

Will return the error message

```
my Str $e;
{   fail CSV::Diag(error => 2034);
    CATCH { default { $e = ~$_; }}
    }
# $e is now "EIF - Loose unescaped quote"
```

list context

All of the 6 items can be retrieved as a list or positional:

```
{   fail CSV::Diag(error => 2034);
    CATCH { default { $_[0].say; }}
    }
```

The indices are chosen to be compatible with the old API

```
$e[0] = error number
$e[1] = error message
$e[2] = error position in buffer
$e[3] = field number
$e[4] = record number
$e[5] = errror buffer
```

hash context

All of the 6 items can be retrieved as a hash entry

```
{   fail CSV::Diag(error => 2034);
    CATCH { default { $_<errno>.say; }}
    }
```

The keys are chosen to be compatible with the old API.

```
$e<errno>  = error number
$e<error>  = error message
$e<pos>    = error position in buffer
$e<field>  = field number
$e<recno>  = record number
$e<buffer> = errror buffer
```

The CSV::Diag is also used by this Text::CSV method

error_diag

error-diag

```
$csv.error_diag;
$error_code = +$csv.error_diag;
$error_str  = ~$csv.error_diag;
($cde, $str, $pos, $rec, $fld) = $csv.error_diag;
```

This function returns the diagnostics of the most recent error.

If called in void context, this will print the internal error code and the associated error message along with the record number, the position of the failure and the buffer of failure with an eject symbol at that position:

```
$csv.parse(q{ "",})
$csv.error_diag;
```

will print

```
EIF - Loose unescaped quote : error 2034 @ record 1, field 1, position 2
 "?",
```

If called in list context, this will return the error code, the error message, the location within the line that was being parsed, the record number, and the buffer itself in that order. Their values are 1-based. The position currently is index of the character at which the parsing failed in the current record. The record number the index of the record parsed by the csv instance. The field number is the index of the field the parser thinks it is currently trying to parse.

If called as `+$csv.error_diag` or `$csv.error_diag.Num`, it will return the error code. If called as `~$csv.error_diag` or `$csv.error_diag.Str` it will return the error message.

record_number

```
$recno = $csv.record_number;
```

Returns the records parsed by this csv instance. This value should be more accurate than `$.` when embedded newlines come in play. Records written by this instance are not counted.

set_diag

```
$csv.set_diag(0);
$csv.set_diag(2025, pos => 12, fieldno => 4, recno => 99);
```

Use to (re)set the diagnostics if you are dealing with errors.

CSV::Field

The fields are internally represented as CSV::Field objects. Any methods that directly or indirectly supports the `meta` attribute controls weather the returned fields will be of this CSV::Field type or that the fields are simplified to a simple basic type.

If the fields are represented/returned as CSV::Field, it supports these methods:

new

```
my CSV::Field $f .= new;
my $f = CSV::Field.new("foo");
my $f = CSV::Field.new(1);
```

Instantiate a new field. Optionally takes a `Cool`.

Bool

```
?$f     and say "The field is true";
$f.Bool and say "This field is true too";
```

Return the boolean value of the field. As CSV is text-only be design, this will also return `False` for `"0"`, where raku sees `"0"` as `True`.

text =head2 Str

```
$str = ~$f;
$str = $f.Str;
$str = $f.text;
```

Return the string representation of the field.

Buf

```
$buf = $f.Buf;
```

Return the field data as "utf8-c8" encoded Buf.

Numeric

```
$i = +$f;
$i = $f.Int;
$i = $f.Num;
```

Return the numeric representation of the field.

gist

```
$f.gist.say;
```

Will show a complete compressed representation of the field with properties. `Q` is quoted, `q` is unquoted. Likewise for `B/b` for binary, `8/7` for Unicode-ness and `M/m` for missing.

A field that was parsed as `,cat,` would return

```
qb7m:"cat"
```

A field parsed as `,"Hëłlº",` would return

```
QB8m:"Ħēłlº"
```

add

```
$f.add($str);
```

Accepts a Str to be added to this field.

set_quoted

```
$f.set_quoted;
```

Set the fact the the field was/is quoted.

is_quoted

```
$f.is_quoted and say "The field was quoted in CSV";
```

Is `True` when the parsed field was quoted in the source.

undefined

```
$f.undefined and say "The field is undefined";
```

Returns `True` when the field is undefined. As CSV is all about strings, the various options that allow interpreting empty fields as undefined make this a required method.

is_binary

WIP: utf8-c8

```
$f.is_binary and say "Do we need a Buf instead?";
```

Returns true if the field has data beyond ASCII.

is_utf8

```
$f.is_utf8 or say "The field is empty or pure ACII";
```

Returns `True` if the field is beyond ASCII, but still valid UTF-8.

is_missing

WIP

```
$f.is_missing and fail;
```

Returns `True` if the field is missing.

CSV::Row

This class is a wrapper over the current row of fields just to add convenience methods.

This is the only argument to callbacks.

The fields in `CSV::Row` are always of class `CSV::Field` and thus contain all meta-information, even if the `Text::CSV`

attribute meta_info is False.

methods

new

```
my CSV::Row $row .= new;
my CSV::Row $row .= new(csv => $csv, fields => @f.map({ CSV::Field.new(*) });
```

csv

The current Text::CSV object related to this row.

elems

Return the number of fields in the row.

fields

The fields (CSV::Field items) this row consist of.

push

```
$row.push(CSV::Field.new(1));
$row.push(1);
$row.push("foo");
$row.push($another-row);
```

Pushing simple things onto the row will extend the row by converting these to CSV::Field objects.

Pushing a CSV::Row onto the row will extend that row with the fields of the row being pushed.

pop

```
my CSV::Field $f = $row.pop;
```

Str

```
my $str = $row.Str;
$io.say(~$row);
```

The stringification of the CSV::Row object is like invoking the string method. This only works if there is a Text::CSV object known to the CSV::Row instance.

hash

```
my %h = $row.hash;
```

Returns the hash with .csv's column_names as keys and the .text of each matching fields entry as values.

strings

```
my @l = $row.strings;
```

Returns the .text part of each entry in .fields.

The row allow direct indexing and iteration as well as hash addressing when "column_names" are set.

```
my $field = $row[1];
my $field = $row<foo>;
```

The last parsed row of a Text::CSV|/Text::CSV> can be acquired using

```
my CSV::Row $row = $csv.row;
```

##

FUNCTIONS

csv

This is an high-level function that aims at simple (user) interfaces. This can be used to read/parse a `csv` file or stream (the default behavior) or to produce a file or write to a stream (define the `out` attribute). It returns an array- or hash-reference on parsing or the numeric value of "error_diag" on writing. When this function fails you can get to the error using the class call to "error_diag"

```
my $aoa = csv(in => "test.csv") or
    die Text::CSV.error_diag;
```

This function takes the arguments as key-value pairs. This can be passed as a list or as an anonymous hash:

```
my $aoa = csv(in => "test.csv", sep => ";");
my $aoh = csv(in => $fh, :headers);
```

The arguments passed consist of two parts: the arguments to "csv" itself and the optional attributes to the `csv` object used inside the function as enumerated and explained in "new".

If not overridden, the default option used for CSV is

```
auto_diag => 1
```

The option that is always set and cannot be altered is

```
binary    => 1
```

in

Used to specify the source. `in` can be a file name (e.g. `"file.csv"`), which will be opened for reading and closed when finished, a file handle (e.g. `$fh` or `FH`), a reference to a glob (e.g. `\*ARGV`), the glob itself (e.g. `*STDIN`), or a reference to a scalar (e.g. `\q{1,2,"csv"}`).

When used with "out", `in` should be a reference to a CSV structure (AoA or AoH) or a Callable (Sub, Routine, Code, or Block) that returns an array-reference or a hash-reference. The code-ref will be invoked with no arguments.

```
my $aoa = csv(in => "file.csv");

open my $fh, "<", "file.csv";
my $aoa = csv(in => $fh);

my $csv = [[qw( Foo Bar )], [ 1, 2 ], [ 2, 3 ]];
my $err = csv(in => $csv, out => "file.csv");
```

The `in` attribute supports a wide range of types, all of which can be combined with the use of `fragment`:

Str

```
csv(in => "file.csv")
```

A plain string is interpreted as a file name to be opened for parsing.

IO

```
my $io = open "file.csv", :r;
csv(in => $io);
```

Parse from the already opened data stream.

Capture

```
csv(in => \("str,ing"));
```

Parse from a single string.

Array of Strings

```
csv(in => ["a,b\n1,2\n3,4\n"]);
csv(in => ["a,b", "1,2", "3,4"]);
```

Parse from the String(s)

Array of Data

```
csv(in => [[<a b>], [1, 2], [3, 4]]);
csv(in => [{:a(1), :b(2)}, {:a(3), :b(4)}]);
```

Use the data as provided

Sub, Routine

```
sub provider {
    @data.elems == 0 and return False;
    return @data.pop;
    }
csv(in => &provider);
```

While the providing Routine returns a data row, use that as is. Stop when the provider returns `False`.

Callable, Block

```
csv(in => { $sth.fetch });
```

While the providing Callable returns a data row, use that as is. Stop when the provider returns `False`.

Supply

```
my $supply = Supply.from-list(@data);
csv(in => $supply);
```

Fetch data rows from the supply.

Channel

```
my $ch = Channel.new;
start {
    $ch.send($_) for @data;
    $ch.close;
    }
csv(in => $ch);
```

Fetch data from the Channel.

Iterator

```
csv(in => @data.iterator);
```

Fetch data rows from the iterator.

Any

```
csv(in => Any);
```

is a shortcut/fallback for

```
csv(in => $*IN);
```

out

In output mode, the default CSV options when producing CSV are

```
eol    => "\r\n"
```

The "fragment" attribute is ignored in output mode.

`out` can be a file name (e.g. `"file.csv"`), which will be opened for writing and closed when finished, a file handle (e.g. `$fh` or `IO::Handle`), a `Channel`, a `Supply`, a `Callable`, or `Nil`.

```
csv(in => sub { $sth.fetch },        out => "dump.csv");
csv(in => { $sth.fetchrow_hashref }, out => "dump.csv",
     headers => $sth.{NAME_lc});
```

When a code-ref is used for `in`, the output is generated per invocation, so no buffering is involved. This implies that there is no size restriction on the number of records. The `csv` function ends when the coderef returns `False`.

When a Callable is used for out, it is called on each row with the row as only argument.

When `Nil` is used for out, the output is completely suppressed. This can be useful when streaming, and the output is not required, but the side-effects of e.g. an `on-in` hook is dealing with the data.

Str:U

```
my $str = csv(in => $in, out => Str);
```

Returns a single String of CSV data.

Str:D

```
my $str = csv(in => $in, out => "out.csv");
```

Writes the data as CSV to the named file.

Array:U

```
my $aoa = csv(in => $in, out => Array);
```

Returns an Array of Arrays.

Hash:U

```
my $aoh = csv(in => $in, out => Hash);
```

Returns an Array of Hashes

IO:D

```
my $io = open "file.csv", :w;
csv(in => $in, out => $io);
```

Writes the data as CSV to the IO handle.

Callable, Block, Sub, Routine

```
my @d;
csv(in => $in, out => { @d.push: $_ });
```

Passes the data rows to the Callable

Channel:U

```
my $ch = csv(in => $in, out => Channel, :!meta);
react {
    whenever $ch -> \row {
        @d.push: row;
        LAST { done; }
        }
    }
```

Writes the data rows into a new Channel, which is returned.

Channel:D

```
my $ch = Channel.new;
my $pr = start {
    react {
        whenever $ch -> \row {
            @d.push: row;
            LAST { done; }
            }
        }
    }
csv(in => $in, out => $ch);
await $pr;
```

Writes the data rows into the existing Channel.

Supplier:D

```
my $sup = Supplier.new;
$sup.Supply.tap (-> \row { @d.push: row; });
csv(in => $in, out => $sup, :!meta);
```

Writes the data rows into the Supplier.

Supply:U

```
my $sup = csv(in => $in, out => Supply, :!meta);
$ch.tap (-> \row { @d.push: row; });
```

Writes the data rows into a new Supply.

encoding

If passed, it should be an encoding accepted by the `:encoding()` option to `open`. There is no default value.

If `encoding` is set to the literal value `"auto"`, the method "header" will be invoked on the opened stream to check if there is a BOM and set the encoding accordingly. This is equal to passing True in the option `detect-bom`.

detect-bom

NYI for various reasons. Also see "is-binary"

If `detect-bom` is given, the method "header" will be invoked on the opened stream to check if there is a BOM and set the encoding accordingly.

`detect_bom` can be abbreviated to `bom`.

This is the same as setting `encoding` to `"auto"`.

Note that as the method "header" is invoked, its default is to also set the headers.

headers

If this attribute is not given, the default behavior is to produce an array of arrays.

If `headers` is supplied, it should be an Array of column names, a Bool, a Hash, a Callable, or a literal flag: `auto`, `lc`, `uc`, or `skip`.

skip

When `skip` is used, the header will not be included in the output.

```
my $aoa = csv(in => $fh, headers => "skip");
```

auto

If `auto` is used, the first line of the `csv` source will be read as the list of field headers and used to produce an array of hashes.

```
my $aoh = csv(in => $fh, headers => "auto");
```

lc

If `lc` is used, the first line of the `csv` source will be read as the list of field headers mapped to lower case and used to produce an array of hashes. This is a variation of `auto`.

```
my $aoh = csv(in => $fh, headers => "lc");
```

uc

If `uc` is used, the first line of the `csv` source will be read as the list of field headers mapped to upper case and used to produce an array of hashes. This is a variation of `auto`.

```
my $aoh = csv(in => $fh, headers => "uc");
```

Bool

If `True` is passed, the method "header" will be invoked with the default options on the opened stream to check if there is a BOM and set the encoding accordingly, detect and set `sep`, `eol` and column names.

Callable

If a Callable is used, the first line of the `csv` source will be read as the list of mangled field headers in which each field is passed as the only argument to the coderef. This list is used to produce an array of hashes.

```
my $i = 0;
my $aoh = csv(in => $fh, headers => { $^h.lc ~ $i++ });
```

this example is a variation of using `lc` where all headers are forced to be unique by adding an index.

ARRAY

If `headers` is an Array, the entries in the list will be used as field names. The first line is considered data instead of headers.

```
my $aoh = csv(in => $fh, headers => [< Foo Bar >]);
```

HASH

If `headers` is a Hash, this implies `auto`, but header fields for that exist as key in the Hash will be replaced by the value for that key. Given a CSV file like

```
post-kode,city,name,id number,fubble
1234AA,Duckstad,Donald,13,"X313DF"
```

using

```
csv (headers => %{ "post-kode" => "pc", "id number" => "ID" }, ...
```

will return an entry like

```
{ pc     => "1234AA",
  city   => "Duckstad",
  name   => "Donald",
  ID     => "13",
  fubble => "X313DF",
  }
```

See also `munge-column-names` and `set-column-names`.

munge-column-names

If `munge-column-names` is set, the method "header" is invoked on the opened stream with all matching arguments to detect and set the headers.

`munge-column-names` can be abbreviated to `munge`.

key

If passed, will default `headers` to `"auto"` and return a hashref instead of an array of hashes. Allowed values are simple strings or arrays where the first element is the joiner and the rest are the fields to join to combine the key

```
my $ref = csv(in => "test.csv", key =>        "code");
my $ref = csv(in => "test.csv", key => [ ":", "code", "color" ]);
```

with test.csv like

```
code,product,price,color
1,pc,850,gray
2,keyboard,12,white
3,mouse,5,black
```

the first example will return

```
{ 1   => {
      code    => 1,
      color   => 'gray',
      price   => 850,
      product => 'pc'
      },
  2   => {
      code    => 2,
      color   => 'white',
      price   => 12,
      product => 'keyboard'
      },
  3   => {
      code    => 3,
      color   => 'black',
      price   => 5,
      product => 'mouse'
      }
  }
```

the second example will return

```
{ "1:gray"    => {
      code    => 1,
      color   => 'gray',
      price   => 850,
      product => 'pc'
      },
  "2:white"   => {
      code    => 2,
      color   => 'white',
      price   => 12,
      product => 'keyboard'
      },
  "3:black"   => {
      code    => 3,
      color   => 'black',
      price   => 5,
      product => 'mouse'
      }
  }
```

fragment

Only output the fragment as defined in the "fragment" method. This option is ignored when *generating* csv. See "out".

Combining all of them could give something like

```
use Text::CSV qw( csv );
my @aoh = csv(
    in       => "test.txt",
    encoding => "utf-8",
    headers  => "auto",
    sep_char => "|",
    fragment => "row=3;6-9;15-*",
    );
say @aoh[15]{Foo};
```

sep-set

If `sep-set` is set, the method "header" is invoked on the opened stream to detect and set `sep` with the given set.

`sep-set` can be abbreviated to `seps`.

Note that as the method "header" is invoked, its default is to also set the headers.

set_column_names

If `set_column_names` is passed, the method "header" is invoked on the opened stream with all arguments meant for "header".

Callbacks

Callbacks enable actions triggered from the *inside* of Text::CSV.

While most of what this enables can easily be done in an unrolled loop as described in the "SYNOPSIS" callbacks, can be used to meet special demands or enhance the "csv" function.

All callbacks except `error` are called with just one argument: the current CSV::Row.

error

```
$csv.callbacks(error => { $csv.SetDiag(0) });
```

the `error` callback is invoked when an error occurs, but *only* when "auto_diag" is set to a true value. This callback is invoked with the values returned by "error_diag":

```
my ($c, $s);

sub ignore3006 (Int $err, Str $msg, Int $pos, Int $recno, Int $fldno) {
    if ($err == 3006) {
        # ignore this error
        ($c, $s) = (Str, Str);
        Text::CSV.SetDiag(0);
        }
    # Any other error
    return;
    } # ignore3006

$csv.callbacks(error => \&ignore3006);
$csv.bind_columns(\$c, \$s);
while ($csv.getline($fh)) {
    # Error 3006 will not stop the loop
    }
```

after_parse

```
sub add-new (CSV::Row $r) { $r.fields.push: "NEW"; }
$csv.callbacks(after_parse => &add-new);
while (my @row = $csv.getline($fh)) {
```

```
    @row[-1] eq "NEW";
    }
```

This callback is invoked after parsing with "getline" only if no error occurred.

The return code of the callback is ignored.

```
sub add_from_db (CSV::Row $r) {
    $sth.execute($r[4]);
    push $r.fields: $sth.fetchrow_array;
    } # add_from_db

my $aoa = csv(in => "file.csv", callbacks => {
    after_parse => &add_from_db });

my $aoa = csv(in => "file.csv", after_parse => {
    $sth.execute($^row[4]); $^row.fields.push: $sth.fetchrow_array; });
```

before_print

```
my $idx = 1;
$csv.callbacks(before_print => { $^row[0] = $idx++ });
$csv.print(*STDOUT, [ 0, $_ ]) for @members;
```

This callback is invoked before printing with "print" only if no error occurred.

The return code of the callback is ignored.

```
sub max_4_fields (CSV::Row $r) {
    $r.elems > 4 and $r.splice (4);
    } # max_4_fields

csv(in => csv(in => "file.csv"), out => *STDOUT,
    callbacks => { before print => \&max_4_fields });

csv(in => csv(in => "file.csv"), out => *STDOUT,
    before print => { $^row.elems > 4 and $^row.splice(4) });
```

This callback is not active for "combine".

Callbacks for csv

The "csv" allows for some callbacks that do not integrate in internals but only feature the "csv" function. XXX: Is this still true?

```
csv(in        => "file.csv",
    callbacks => {
        after_parse  => { say "AFTER PARSE";  }, # first
        after_in     => { say "AFTER IN";     }, # second
        on_in        => { say "ON IN";        }, # third
        },
    );

csv(in        => $aoh,
    out       => "file.csv",
    callbacks => {
        on_in        => { say "ON IN";        }, # first
        before_out   => { say "BEFORE OUT";   }, # second
        before_print => { say "BEFORE PRINT"; }, # third
        },
    );
```

filter

This callback can be used to filter records. It is called just after a new record has been scanned. The callback will be invoked with the current "CSV::Row" and should return `True` for records to accept and `False` for records to reject.

```
csv (in => "file.csv", filter => {
           $^row[2] ~~ /a/ &&  # third field should contain an "a"
           $^row[4].chars > 4  # length of the 5th field minimal 5
           });
```

```
csv (in => "file.csv", filter => "not_blank");
csv (in => "file.csv", filter => "not_empty");
csv (in => "file.csv", filter => "filled");
```

If the filter is used to *alter* the content of a field, make sure that the sub, block or callable returns true in order not to have that record skipped:

```
filter => { $^row[1].text .= uc }
```

will upper-case the second field, and then skip it if the resulting content evaluates to false. To always accept, end with truth:

```
filter => { $^row[1].text .= uc; 1 }}
```

Predefined filters

Given a file like (line numbers prefixed for doc purpose only):

```
1:1,2,3
2:
3:,
4:""
5:,,
6:, ,
7:"",
8:" "
9:4,5,6
```

not_blank =item not-blank

Filter out the blank lines

This filter is a shortcut for

```
filter => { $^row.elems > 1 or
            $^row[0].defined && $^row[0] ne "" or
            $^row[0].is-quoted }
```

With the given example, line 2 will be skipped.

not_empty =item not-empty

Filter out lines where all the fields are empty.

This filter is a shortcut for

```
filter => { $^row.first: { .defined && $_ ne ""   }}
```

A space is not regarded being empty, so given the example data, lines 2, 3, 4, 5, and 7 are skipped.

filled

Filter out lines that have no visible data

This filter is a shortcut for

```
filter => { $^row.first: { .defined && $_ ~~ /\S/ }}
```

This filter rejects all lines that *not* have at least one field that does not evaluate to the empty string.

With the given example data, this filter would skip lines 2 through 8.

after_in

This callback is invoked for each record after all records have been parsed but before returning the

reference to the caller.

This callback can also be passed as an attribute to "csv" without the `callbacks` wrapper.

before_out

This callback is invoked for each record before the record is printed.

This callback can also be passed as an attribute to "csv" without the `callbacks` wrapper.

on_in

This callback acts exactly as the "after_in" or the "before_out" hooks.

This callback can also be passed as an attribute to "csv" without the `callbacks` wrapper.

EXAMPLES

Reading a CSV file line by line:

```
my $csv = Text::CSV.new(:auto_diag);
my $fh = open "file.csv", :r, :!chomp;
while (my @row = $csv.getline($fh)) {
    # do something with @$row
    }
$fh.close;
```

Reading only a single column

```
my $csv = Text::CSV.new(:auto_diag);
my $fh = open "file.csv", :r, :!chomp;
# get only the 4th column
my @column = $csv.getline_all($fh).map(*[3]);
$fh.close;
```

with "csv", you could do

```
my @column = csv(in => "file.csv", fragment => "col=4").map(*[0]);
```

or

```
my @column = csv(in => "file.csv", fragment => "col=4").map(*.flat);
```

Parsing CSV strings:

```
my $csv = Text::CSV.new(:keep_meta);

my $sample_input_string =
    q{"I said, ""Hi!""",Yes,"",2.34,,"1.09","\x[20ac]",};
if ($csv.parse($sample_input_string)) {
    my @field = $csv.fields;
    for ^@field.elems -> $col {
        my $quo = $csv.is_quoted($col) ? $csv.{quote_char} : "";
        printf "%2d: %s%s%s\n", $col, $quo, $field[$col], $quo;
        }
    }
else {
    print STDERR "parse failed on argument: ",
        $csv.error_input, "\n";
    $csv.error_diag;
    }
```

Printing CSV data

The fast way: using "print"

An example for creating csv files using the "print" method:

```
my $csv = Text::CSV.new(eol => $*OUT.nl);
open my $fh, ">", "foo.csv" or die "foo.csv: $!";
for 1..10 -> $x {
    $csv.print($fh, [ $x, ~$x ]) or $csv.error_diag;
    }
close $fh or die "$tbl.csv: $!";
```

The slow way: using "combine" and "string"

or using the slower "combine" and "string" methods:

```
my $csv = Text::CSV.new;

open my $csv_fh, ">", "hello.csv" or die "hello.csv: $!";

my @sample_input_fields = (
    'You said, "Hello!"',   5.67,
    '"Surely"',   '',   '3.14159');
if ($csv.combine(@sample_input_fields)) {
    print $csv_fh $csv.string, "\n";
    }
else {
    print "combine failed on argument: ",
        $csv.error_input, "\n";
    }
close $csv_fh or die "hello.csv: $!";
```

Rewriting CSV

Rewrite csv files with ; as separator character to well-formed csv:

```
use Text::CSV qw( csv );
csv(in => csv(in => "bad.csv", sep_char => ";"), out => *STDOUT);
```

Dumping database tables to CSV

Dumping a database table can be simple as this (TIMTOWTDI):

```
my $dbh = DBI.connect(...);
my $sql = "select * from foo";

# using your own loop
open my $fh, ">", "foo.csv" or die "foo.csv: $!\n";
my $csv = Text::CSV.new(eol => "\r\n");
my $sth = $dbh.prepare($sql); $sth.execute;
$csv.print($fh, $sth.{NAME_lc});
while (my $row = $sth.fetch) {
    $csv.print($fh, $row);
    }

# using the csv function, all in memory
csv(out => "foo.csv", in => $dbh.selectall_arrayref($sql));

# using the csv function, streaming with callbacks
my $sth = $dbh.prepare($sql); $sth.execute;
csv(out => "foo.csv", in => { $sth.fetch            });
csv(out => "foo.csv", in => { $sth.fetchrow_hashref });
```

Note that this does not discriminate between "empty" values and NULL-values from the database, as both will be the same empty field in CSV. To enable distinction between the two, use quote_empty.

```
csv(out => "foo.csv", in => { $sth.fetch }, :quote_empty);
```

If the database import utility supports special sequences to insert NULL values into the database, like MySQL/MariaDB supports \N, use a filter or a map

```
csv(out => "foo.csv", in => { $sth.fetch },
                  on_in => { $_ //= "\\N" for @$_[1] }); # WIP

while (my @row = $sth.fetch) {
    $csv.print($fh, @row.map({ * // "\\N" }));
    }
```

these special sequences are not recognized by Text::CSV_XS on parsing the CSV generated like this, but map and filter are your friends again

```
while (my @row = $csv.getline($io)) {
    $sth.execute(@row.map({ $_ eq "\\N" ?? Nil !! $_ }));
    }

csv(in => "foo.csv", filter => { 1 => {
    $sth.execute(@{$_[1]}.map({ $_ eq "\\N" ?? Nil !! $_ }); False; }});
```

The examples folder

For more extended examples, see the *examples/* [1]) sub-directory in the original distribution or the git repository [2]).

1. https://github.com/Tux/Text-CSV_XS/tree/master/examples/
2. https://github.com/Tux/Text-CSV_XS/

The following files can be found there:

csv-check

This is a command-line tool to check the csv file and report on its content.

TODO

csv2xls

A script to convert csv to Microsoft Excel.

TODO

csvdiff

A script that provides colorized diff on sorted CSV files, assuming first line is header and first field is the key. Output options include colorized ANSI escape codes or HTML.

TODO

##

CAVEATS

Microsoft Excel

The import/export from Microsoft Excel is a *risky task*, according to the documentation in Text::CSV::Separator. Microsoft uses the system's list separator defined in the regional settings, which happens to be a semicolon for Dutch, German and Spanish (and probably some others as well). For the English locale, the default is a comma. In Windows however, the user is free to choose a predefined locale, and then change *every* individual setting in it, so checking the locale is no solution.

A lone first line with just

```
sep=;
```

will be recognized and honored: it will set `sep` to `;` and skip that line.

TODO / WIP / NYI

Real binary data

> The solution would be a working `utf8-c8` encoding.

BOM detection

> There is no working solution yet for detection of BOM on the "header" method. Besides that, not all encodings are supported in raku.

on-in and before-print callbacks

> The "on-in" callback currently is an alias for "after-parse" if the latter is not specified.

Examples

> Convert the perl5 example/tool files to raku versions

Metadata and CSV for the web

> Metadata Vocabulary for Tabular Data (a W3C editor's draft) could be an example for supporting more metadata.
>
> W3C's work CSV on the Web: Use Cases and Requirements is almost finished and worth looking at.

Cookbook

> Write a document that has recipes for most known non-standard (and maybe some standard) `CSV` formats, including formats that use `TAB`, `;`, `|`, or other non-comma separators.
>
> Examples could be taken from W3C's CSV on the Web: Use Cases and Requirements

DIAGNOSTICS

Still under construction ...

This section describes the error codes that are used in perl5's module Text::CSV_XS, and several of these errors are either not applicable in raku or changed slightly. Once all of the API is finished, this section will be cleaned up. The intention of the error coded however remains.

If an error occurs, `$csv.error_diag` can be used to get information on the cause of the failure. Note that for speed reasons the internal value is never cleared on success, so using the value returned by "error_diag" in normal cases - when no error occurred - may cause unexpected results.

If the constructor failed, the cause will be thrown as an Exception that represents "error_diag".

The `$csv.error_diag` method is automatically invoked upon error when the contractor was called with `auto_diag` set to `True`.

Errors can be (individually) caught using the "error" callback.

The errors as described below are available. I have tried to make the error itself explanatory enough, but more descriptions will be added. For most of these errors, the first three capitals describe the error category:

- INI

  Initialization error or option conflict.

- ECR

  Carriage-Return related parse error.

- EOF

  End-Of-File related parse error.

- EIQ

  Parse error inside quotation.

- EIF

  Parse error inside field.

- ECB

  Combine error.

- EHR

  Hash parse related error.

- EHK

  Errors related to hooks/callbacks.

- CSV

  Errors related to the csv function.

And below should be the complete list of error codes that can be returned:

- 1001 "INI - separator is equal to quote- or escape sequence"

  The separation sequence cannot be equal to the quotation sequence or to the escape sequence, as this would invalidate all parsing rules.

- 1002 "INI - allow_whitespace with escape_char or quote_char SP or TAB"

  Using the `allow_whitespace` attribute when either `quote_char` or `escape_char` is equal to `SPACE` or `TAB` is too ambiguous to allow.

- 1003 "INI - \r or \n in main attr not allowed"

  Using default `eol` sequences in either separation sequence, quotation sequence, or escape sequence is not allowed.

- 1004 "INI - callbacks should be undefined or a hashref"

  The `callbacks` attribute only allows one to be undefined or a hash reference.

- 1010 "INI - the header is empty"

  The header line parsed in the "header" is empty.

- 1011 "INI - the header contains more than one valid separator"

  The header line parsed in the "header" contains more than one (unique) separator character out of the allowed set of separators.

- 1012 "INI - the header contains an empty field"

  The header line parsed in the "header" is contains an empty field.

- 1013 "INI - the header contains nun-unique fields"

  The header line parsed in the "header" contains at least two identical fields.

- 2010 "ECR - QUO char inside quotes followed by CR not part of EOL"

  When `eol` has been set to anything but the default, like `"\r\t\n"`, and the `"\r"` is following the **second** (closing) `quote_char`, where the characters following the `"\r"` do not make up the `eol` sequence, this is an error.

- 2011 "ECR - Characters after end of quoted field"

  Sequences like `1,foo,"bar"baz,22,1` are not allowed. `"bar"` is a quoted field and after the closing double-quote, there should be either a new-line sequence or a separation sequence.

- 2012 "EOF - End of data in parsing input stream"

  Self-explaining. End-of-file while inside parsing a stream. Can happen only when reading from streams with "getline", as using "parse" is done on strings that are not required to have a trailing `eol`.

- 2013 "INI - Specification error for fragments RFC7111"

  Invalid specification for URI "fragment" specification.

- 2021 "EIQ - NL char inside quotes, binary off"

  Sequences like `1,"foo\nbar",22,1` are allowed only when the binary option has been selected with the constructor.

- 2022 "EIQ - CR char inside quotes, binary off"

  Sequences like `1,"foo\rbar",22,1` are allowed only when the binary option has been selected with the constructor.

- 2023 "EIQ - QUO sequence not allowed"

  Sequences like `"foo "bar" baz",qu` and `2023,",2008-04-05,"Foo, Bar",\n` will cause this error.

- 2024 "EIQ - EOF cannot be escaped, not even inside quotes"

  The escape sequence is not allowed as last item in an input stream.

- 2025 "EIQ - Loose unescaped escape"

  An escape sequence should escape only characters that need escaping.

Allowing the escape for other characters is possible with the attribute "allow_loose_escapes".

- 2026 "EIQ - Binary character inside quoted field, binary off"

  Binary characters are not allowed by default. Exceptions are fields that contain valid UTF-8, that will automatically be upgraded if the content is valid UTF-8. Set `binary` to `1` to accept binary data.

- 2027 "EIQ - Quoted field not terminated"

  When parsing a field that started with a quotation sequence, the field is expected to be closed with a quotation sequence. When the parsed line is exhausted before the quote is found, that field is not terminated.

- 2030 "EIF - NL char inside unquoted verbatim, binary off"
- 2031 "EIF - CR char is first char of field, not part of EOL"
- 2032 "EIF - CR char inside unquoted, not part of EOL"
- 2034 "EIF - Loose unescaped quote"
- 2035 "EIF - Escaped EOF in unquoted field"
- 2036 "EIF - ESC error"
- 2037 "EIF - Binary character in unquoted field, binary off"
- 2110 "ECB - Binary character in Combine, binary off"
- 2200 "EIO - print to IO failed. See errno"
- 3001 "EHR - Unsupported syntax for column_names"
- 3002 "EHR - getline_hr called before column_names"
- 3003 "EHR - bind_columns and column_names fields count mismatch"
- 3004 "EHR - bind_columns only accepts refs to scalars"
- 3006 "EHR - bind_columns did not pass enough refs for parsed fields"
- 3007 "EHR - bind_columns needs refs to writable scalars"
- 3008 "EHR - unexpected error in bound fields"
- 3009 "EHR - print_hr called before column_names"
- 3010 "EHR - print_hr called with invalid arguments"
- 3100 "EHK - Unsupported callback"
- 4001 "PRM - The key does not exist as field in the data",

  You cannot set the key from a non-existing column.

  If you were using a key list, *all* keys should exist.

- 5000 "CSV - Unsupported type for in"
- 5001 "CSV - Unsupported type for out"

SEE ALSO

Modules in perl5:

IO::File, IO::Handle, IO::Wrap, Text::CSV_XS, Text::CSV, Text::CSV_PP, Text::CSV::Encoded, Text::CSV::Separator, Text::CSV::Slurp, Spreadsheet::CSV and Spreadsheet::Read;

AUTHOR

H.Merijn Brand <*h.m.brand@xs4all.nl*> wrote this based on the features provided by perl5's Text::CSV_XS.

Liz Mattijsen helped in getting the best out of raku.

COPYRIGHT AND LICENSE