The so-called CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases. CSV format was used for many years prior to attempts to describe the format in a standardized way in RFC 4180. The lack of a well-defined standard means that subtle differences often exist in the data produced and consumed by different applications. These differences can make it annoying to process CSV files from multiple sources. Still, while the delimiters and quoting characters vary, the overall format is similar enough that it is possible to write a single module which can efficiently manipulate such data, hiding the details of reading and writing the data from the programmer.

The csv module implements classes to read and write tabular data in CSV format. It allows programmers to say, “write this data in the format preferred by Excel,” or “read data from this file which was generated by Excel,” without knowing the precise details of the CSV format used by Excel. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats.

The csv module’s reader and writer objects read and write sequences. Programmers can also read and write data in dictionary form using the DictReader and DictWriter classes.

See also:

PEP 305 - CSV File APIThe Python Enhancement Proposal which proposed this addition to Python.

14.1.1. Module Contents

The csv module defines the following functions:

csv.reader(csvfile, dialect='excel', \*\*fmtparams)

Return a reader object which will iterate over lines in the given csvfile. csvfile can be any object which supports the iterator protocol and returns a string each time its \_\_next\_\_() method is called — file objects and list objects are both suitable. If csvfile is a file object, it should be opened with newline=''. [1] An optional dialect parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the Dialect class or one of the strings returned by the list\_dialects() function. The other optional fmtparams keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about the dialect and formatting parameters, see section Dialects and Formatting Parameters.

Each row read from the csv file is returned as a list of strings. No automatic data type conversion is performed unless the QUOTE\_NONNUMERIC format option is specified (in which case unquoted fields are transformed into floats).

A short usage example:

>>>>>> import csv

>>> with open('eggs.csv', newline='') as csvfile:

... spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')

... for row in spamreader:

... print(', '.join(row))

Spam, Spam, Spam, Spam, Spam, Baked Beans

Spam, Lovely Spam, Wonderful Spam

csv.writer(csvfile, dialect='excel', \*\*fmtparams)

Return a writer object responsible for converting the user’s data into delimited strings on the given file-like object. csvfile can be any object with a write() method. If csvfile is a file object, it should be opened with newline='' [1]. An optional dialect parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the Dialect class or one of the strings returned by the list\_dialects() function. The other optional fmtparams keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about the dialect and formatting parameters, see section Dialects and Formatting Parameters. To make it as easy as possible to interface with modules which implement the DB API, the value None is written as the empty string. While this isn’t a reversible transformation, it makes it easier to dump SQL NULL data values to CSV files without preprocessing the data returned from a cursor.fetch\* call. All other non-string data are stringified with str() before being written.

A short usage example:

import csv

with open('eggs.csv', 'w', newline='') as csvfile:

spamwriter = csv.writer(csvfile, delimiter=' ',

quotechar='|', quoting=csv.QUOTE\_MINIMAL)

spamwriter.writerow(['Spam'] \* 5 + ['Baked Beans'])

spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])

csv.register\_dialect(name, [dialect, ]\*\*fmtparams)

Associate dialect with name. name must be a string. The dialect can be specified either by passing a sub-class of Dialect, or by fmtparams keyword arguments, or both, with keyword arguments overriding parameters of the dialect. For full details about the dialect and formatting parameters, see section Dialects and Formatting Parameters.

csv.unregister\_dialect(name)

Delete the dialect associated with name from the dialect registry. An Error is raised if name is not a registered dialect name.

csv.get\_dialect(name)

Return the dialect associated with name. An Error is raised if name is not a registered dialect name. This function returns an immutable Dialect.

csv.list\_dialects()

Return the names of all registered dialects.

csv.field\_size\_limit([new\_limit])

Returns the current maximum field size allowed by the parser. If new\_limit is given, this becomes the new limit.

The csv module defines the following classes:

class csv.DictReader(csvfile, fieldnames=None, restkey=None, restval=None, dialect='excel', \*args, \*\*kwds)

Create an object which operates like a regular reader but maps the information read into a dict whose keys are given by the optional fieldnames parameter. The fieldnames parameter is a sequence whose elements are associated with the fields of the input data in order. These elements become the keys of the resulting dictionary. If the fieldnames parameter is omitted, the values in the first row of the csvfile will be used as the fieldnames. If the row read has more fields than the fieldnames sequence, the remaining data is added as a sequence keyed by the value of restkey. If the row read has fewer fields than the fieldnames sequence, the remaining keys take the value of the optional restval parameter. Any other optional or keyword arguments are passed to the underlying reader instance.

class csv.DictWriter(csvfile, fieldnames, restval='', extrasaction='raise', dialect='excel', \*args, \*\*kwds)

Create an object which operates like a regular writer but maps dictionaries onto output rows. The fieldnames parameter is a sequence of keys that identify the order in which values in the dictionary passed to the writerow() method are written to the csvfile. The optional restval parameter specifies the value to be written if the dictionary is missing a key in fieldnames. If the dictionary passed to the writerow() method contains a key not found in fieldnames, the optional extrasaction parameter indicates what action to take. If it is set to 'raise' a ValueError is raised. If it is set to 'ignore', extra values in the dictionary are ignored. Any other optional or keyword arguments are passed to the underlying writer instance.

Note that unlike the DictReader class, the fieldnames parameter of the DictWriter is not optional. Since Python’s dict objects are not ordered, there is not enough information available to deduce the order in which the row should be written to the csvfile.

class csv.Dialect

The Dialect class is a container class relied on primarily for its attributes, which are used to define the parameters for a specific reader or writer instance.

class csv.excel

The excel class defines the usual properties of an Excel-generated CSV file. It is registered with the dialect name 'excel'.

class csv.excel\_tab

The excel\_tab class defines the usual properties of an Excel-generated TAB-delimited file. It is registered with the dialect name 'excel-tab'.

class csv.unix\_dialect

The unix\_dialect class defines the usual properties of a CSV file generated on UNIX systems, i.e. using '\n' as line terminator and quoting all fields. It is registered with the dialect name 'unix'.

New in version 3.2.

class csv.Sniffer

The Sniffer class is used to deduce the format of a CSV file.

The Sniffer class provides two methods:

sniff(sample, delimiters=None)

Analyze the given sample and return a Dialect subclass reflecting the parameters found. If the optional delimiters parameter is given, it is interpreted as a string containing possible valid delimiter characters.

has\_header(sample)

Analyze the sample text (presumed to be in CSV format) and return True if the first row appears to be a series of column headers.

An example for Sniffer use:

with open('example.csv') as csvfile:

dialect = csv.Sniffer().sniff(csvfile.read(1024))

csvfile.seek(0)

reader = csv.reader(csvfile, dialect)

# ... process CSV file contents here ...

The csv module defines the following constants:

csv.QUOTE\_ALL

Instructs writer objects to quote all fields.

csv.QUOTE\_MINIMAL

Instructs writer objects to only quote those fields which contain special characters such as delimiter, quotechar or any of the characters in lineterminator.

csv.QUOTE\_NONNUMERIC

Instructs writer objects to quote all non-numeric fields.

Instructs the reader to convert all non-quoted fields to type float.

csv.QUOTE\_NONE

Instructs writer objects to never quote fields. When the current delimiter occurs in output data it is preceded by the current escapechar character. If escapechar is not set, the writer will raise Error if any characters that require escaping are encountered.

Instructs reader to perform no special processing of quote characters.

The csv module defines the following exception:

exception csv.Error

Raised by any of the functions when an error is detected.

14.1.2. Dialects and Formatting Parameters

To make it easier to specify the format of input and output records, specific formatting parameters are grouped together into dialects. A dialect is a subclass of the Dialect class having a set of specific methods and a single validate() method. When creating reader or writer objects, the programmer can specify a string or a subclass of the Dialect class as the dialect parameter. In addition to, or instead of, the dialect parameter, the programmer can also specify individual formatting parameters, which have the same names as the attributes defined below for the Dialect class.

Dialects support the following attributes:

Dialect.delimiter

A one-character string used to separate fields. It defaults to ','.

Dialect.doublequote

Controls how instances of quotechar appearing inside a field should be themselves be quoted. When True, the character is doubled. When False, the escapechar is used as a prefix to the quotechar. It defaults to True.

On output, if doublequote is False and no escapechar is set, Error is raised if a quotechar is found in a field.

Dialect.escapechar

A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE\_NONE and the quotechar if doublequote is False. On reading, the escapechar removes any special meaning from the following character. It defaults to None, which disables escaping.

Dialect.lineterminator

The string used to terminate lines produced by the writer. It defaults to '\r\n'.

Note:

The reader is hard-coded to recognise either '\r' or '\n' as end-of-line, and ignores lineterminator. This behavior may change in the future.

Dialect.quotechar

A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '"'.

Dialect.quoting

Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE\_\* constants (see section Module Contents) and defaults to QUOTE\_MINIMAL.

Dialect.skipinitialspace

When True, whitespace immediately following the delimiter is ignored. The default is False.

Dialect.strict

When True, raise exception Error on bad CSV input. The default is False.

14.1.3. Reader Objects

Reader objects (DictReader instances and objects returned by the reader() function) have the following public methods:

csvreader.\_\_next\_\_()

Return the next row of the reader’s iterable object as a list, parsed according to the current dialect. Usually you should call this as next(reader).

Reader objects have the following public attributes:

csvreader.dialect

A read-only description of the dialect in use by the parser.

csvreader.line\_num

The number of lines read from the source iterator. This is not the same as the number of records returned, as records can span multiple lines.

DictReader objects have the following public attribute:

csvreader.fieldnames

If not passed as a parameter when creating the object, this attribute is initialized upon first access or when the first record is read from the file.

14.1.4. Writer Objects

Writer objects (DictWriter instances and objects returned by the writer() function) have the following public methods. A row must be a sequence of strings or numbers for Writer objects and a dictionary mapping fieldnames to strings or numbers (by passing them through str() first) for DictWriter objects. Note that complex numbers are written out surrounded by parens. This may cause some problems for other programs which read CSV files (assuming they support complex numbers at all).

csvwriter.writerow(row)

Write the row parameter to the writer’s file object, formatted according to the current dialect.

csvwriter.writerows(rows)

Write all the rows parameters (a list of row objects as described above) to the writer’s file object, formatted according to the current dialect.

Writer objects have the following public attribute:

csvwriter.dialect

A read-only description of the dialect in use by the writer.

DictWriter objects have the following public method:

DictWriter.writeheader()

Write a row with the field names (as specified in the constructor).

New in version 3.2.

14.1.5. Examples

The simplest example of reading a CSV file:

import csv

with open('some.csv', newline='') as f:

reader = csv.reader(f)

for row in reader:

print(row)

Reading a file with an alternate format:

import csv

with open('passwd', newline='') as f:

reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE\_NONE)

for row in reader:

print(row)

The corresponding simplest possible writing example is:

import csv

with open('some.csv', 'w', newline='') as f:

writer = csv.writer(f)

writer.writerows(someiterable)

Since open() is used to open a CSV file for reading, the file will by default be decoded into unicode using the system default encoding (see locale.getpreferredencoding()). To decode a file using a different encoding, use the encoding argument of open:

import csv

with open('some.csv', newline='', encoding='utf-8') as f:

reader = csv.reader(f)

for row in reader:

print(row)

The same applies to writing in something other than the system default encoding: specify the encoding argument when opening the output file.

Registering a new dialect:

import csv

csv.register\_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE\_NONE)

with open('passwd', newline='') as f:

reader = csv.reader(f, 'unixpwd')

A slightly more advanced use of the reader — catching and reporting errors:

import csv, sys

filename = 'some.csv'

with open(filename, newline='') as f:

reader = csv.reader(f)

try:

for row in reader:

print(row)

except csv.Error as e:

sys.exit('file {}, line {}: {}'.format(filename, reader.line\_num, e))

And while the module doesn’t directly support parsing strings, it can easily be done:

import csv

for row in csv.reader(['one,two,three']):

print(row)

Footnotes

[1] (1, 2) If newline='' is not specified, newlines embedded inside quoted fields will not be interpreted correctly, and on platforms that use \r\n linendings on write an extra \r will be added. It should always be safe to specify newline='', since the csv module does its own (universal) newline handling.