13.1. csv — CSV File Reading and Writing

New in version 2.3.

The so-called CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases. There is no "CSV standard", so the format is operationally defined by the many applications which read and write it. The lack of a 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.

Note: This version of the `csv` module doesn't support Unicode input. Also, there are currently some issues regarding ASCII NUL characters. Accordingly, all input should be UTF-8 or printable ASCII to be safe; see the examples in section Examples.

See also:

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

13.1.1. 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 must be opened with the 'b' flag on platforms where that makes a difference. 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.

A short usage example:

```
>>> import csv
>>> with open('eggs.csv', 'rb') 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
```

Changed in version 2.5: The parser is now stricter with respect to multi-line quoted fields. Previously, if a line ended within a quoted field without a terminating newline character, a newline would be inserted into the returned field. This behavior caused problems when reading files which contained carriage return characters within fields. The behavior was changed to return the field without inserting newlines. As a consequence, if newlines embedded within fields are important, the input should be split into lines in a manner which preserves the newline characters.

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 must be opened with the 'b' flag on platforms where that makes a difference. 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. Floats are stringified with `repr()` before being written. All other non-string data are stringified with `str()` before being written.

A short usage example:

```
import csv
with open('eggs.csv', 'wb') 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 or Unicode object. 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.

Changed in version 2.5: This function now returns an immutable `Dialect`. Previously an instance of the requested dialect was returned. Users could modify the underlying class, changing the behavior of active readers and writers.

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.

New in version 2.5.

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.

A short usage example:

```
>>> import csv
>>> with open('names.csv') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Baked Beans
Lovely Spam
Wonderful Spam
```

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*.

A short usage example:

```
import csv

with open('names.csv', 'w') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

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. 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', 'rb') 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.

13.1.2. 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 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`.

13.1.3. 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.

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.

New in version 2.5.

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.

Changed in version 2.6.

13.1.4. 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 2.7.

13.1.5. Examples

The simplest example of reading a CSV file:

```
import csv
with open('some.csv', 'rb') as f:
    reader = csv.reader(f)
    for row in reader:
        print row
```

Reading a file with an alternate format:

```
import csv
with open('passwd', 'rb') 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', 'wb') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

Registering a new dialect:

```
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', 'rb') 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, 'rb') as f:
    reader = csv.reader(f)
```

```
    try:
        for row in reader:
            print row
    except csv.Error as e:
        sys.exit('file %s, line %d: %s' % (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
```

The `csv` module doesn't directly support reading and writing Unicode, but it is 8-bit-clean save for some problems with ASCII NUL characters. So you can write functions or classes that handle the encoding and decoding for you as long as you avoid encodings like UTF-16 that use NULs. UTF-8 is recommended.

`unicode_csv_reader()` below is a generator that wraps `csv.reader` to handle Unicode CSV data (a list of Unicode strings). `utf_8_encoder()` is a generator that encodes the Unicode strings as UTF-8, one string (or row) at a time. The encoded strings are parsed by the CSV reader, and `unicode_csv_reader()` decodes the UTF-8-encoded cells back into Unicode:

```
import csv

def unicode_csv_reader(unicode_csv_data, dialect=csv.excel, **kwargs):
    # csv.py doesn't do Unicode; encode temporarily as UTF-8:
    csv_reader = csv.reader(utf_8_encoder(unicode_csv_data),
                            dialect=dialect, **kwargs)
    for row in csv_reader:
        # decode UTF-8 back to Unicode, cell by cell:
        yield [unicode(cell, 'utf-8') for cell in row]

def utf_8_encoder(unicode_csv_data):
    for line in unicode_csv_data:
        yield line.encode('utf-8')
```

For all other encodings the following `UnicodeReader` and `UnicodeWriter` classes can be used. They take an additional *encoding* parameter in their constructor and make sure that the data passes the real reader or writer encoded as UTF-8:

```
import csv, codecs, cStringIO

class UTF8Recoder:
    """
    Iterator that reads an encoded stream and reencodes the input to UTF-8
    """
    def __init__(self, f, encoding):
        self.reader = codecs.getreader(encoding)(f)

    def __iter__(self):
        return self

    def next(self):
        return self.reader.next().encode("utf-8")

class UnicodeReader:
    """
    A CSV reader which will iterate over lines in the CSV file "f",
    which is encoded in the given encoding.
    """

    def __init__(self, f, dialect=csv.excel, encoding="utf-8", **kwds):
        f = UTF8Recoder(f, encoding)
        self.reader = csv.reader(f, dialect=dialect, **kwds)

    def next(self):
        row = self.reader.next()
        return [unicode(s, "utf-8") for s in row]

    def __iter__(self):
        return self

class UnicodeWriter:
    """
    A CSV writer which will write rows to CSV file "f",
    which is encoded in the given encoding.
    """
```

```
    def __init__(self, f, dialect=csv.excel, encoding="utf-8", **kwds):
        # Redirect output to a queue
        self.queue = cStringIO.StringIO()
        self.writer = csv.writer(self.queue, dialect=dialect, **kwds)
        self.stream = f
        self.encoder = codecs.getincrementalencoder(encoding)()

    def writerow(self, row):
        self.writer.writerow([s.encode("utf-8") for s in row])
        # Fetch UTF-8 output from the queue ...
        data = self.queue.getvalue()
        data = data.decode("utf-8")
        # ... and reencode it into the target encoding
        data = self.encoder.encode(data)
        # write to the target stream
        self.stream.write(data)
        # empty queue
        self.queue.truncate(0)

    def writerows(self, rows):
        for row in rows:
            self.writerow(row)
```