csv – Comma-separated value files

Purpose:	Read and write comma separated value files.
Available In:	2.3 and later

The `csv` module is useful for working with data exported from spreadsheets and databases into text files formatted with fields and records, commonly referred to as *comma-separated value* (CSV) format because commas are often used to separate the fields in a record.

Note: The Python 2.5 version of `csv` does not support Unicode data. There are also "issues with ASCII NUL characters". Using UTF-8 or printable ASCII is recommended.

Reading

Use `reader()` to create a an object for reading data from a CSV file. The reader can be used as an iterator to process the rows of the file in order. For example:

```
import csv
import sys

f = open(sys.argv[1], 'rt')
try:
    reader = csv.reader(f)
    for row in reader:
        print row
finally:
    f.close()
```

The first argument to `reader()` is the source of text lines. In this case, it is a file, but any iterable is accepted (`StringIO` instances, lists, etc.). Other optional arguments can be given to control how the input data is parsed.

This example file was exported from NeoOffice.

```
"Title 1","Title 2","Title 3"
1,"a",08/18/07
2,"b",08/19/07
3,"c",08/20/07
4,"d",08/21/07
5,"e",08/22/07
6,"f",08/23/07
7,"g",08/24/07
8,"h",08/25/07
9,"i",08/26/07
```

As it is read, each row of the input data is parsed and converted to a list of strings.

```
$ python csv_reader.py testdata.csv

['Title 1', 'Title 2', 'Title 3']
['1', 'a', '08/18/07']
['2', 'b', '08/19/07']
['3', 'c', '08/20/07']
['4', 'd', '08/21/07']
['5', 'e', '08/22/07']
['6', 'f', '08/23/07']
['7', 'g', '08/24/07']
['8', 'h', '08/25/07']
['9', 'i', '08/26/07']
```

The parser handles line breaks embedded within strings in a row, which is why a "row" is not always the same as a "line" of input from the file.

```
"Title 1","Title 2","Title 3"
1,"first line
second line",08/18/07
```

Values with line breaks in the input retain the internal line breaks when returned by the parser.

Navigation

Table of Contents
Previous: File Formats
Next: ConfigParser – Work with configuration files

This Page

Show Source

Examples

The output from all the example programs from PyMOTW has been generated with Python 2.7.8, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

If you are looking for examples that work under Python 3, please refer to the PyMOTW-3 section of the site.

```
$ python csv_reader.py testlinebreak.csv

['Title 1', 'Title 2', 'Title 3']
['1', 'first line\nsecond line', '08/18/07']
```

Writing

Writing CSV files is just as easy as reading them. Use **`writer()`** to create an object for writing, then iterate over the rows, using **`writerow()`** to print them.

```
import csv
import sys

f = open(sys.argv[1], 'wt')
try:
    writer = csv.writer(f)
    writer.writerow( ('Title 1', 'Title 2', 'Title 3') )
    for i in range(10):
        writer.writerow( (i+1, chr(ord('a') + i), '08/%02d/07'
finally:
    f.close()

print open(sys.argv[1], 'rt').read()
```

The output does not look exactly like the exported data used in the reader example:

```
$ python csv_writer.py testout.csv

Title 1,Title 2,Title 3
1,a,08/01/07
2,b,08/02/07
3,c,08/03/07
4,d,08/04/07
5,e,08/05/07
6,f,08/06/07
7,g,08/07/07
8,h,08/08/07
9,i,08/09/07
10,j,08/10/07
```

The default quoting behavior is different for the writer, so the string column is not quoted. That is easy to change by adding a quoting argument to quote non-numeric values:

```
writer = csv.writer(f, quoting=csv.QUOTE_NONNUMERIC)
```

And now the strings are quoted:

```
$ python csv_writer_quoted.py testout_quoted.csv

"Title 1","Title 2","Title 3"
1,"a","08/01/07"
2,"b","08/02/07"
3,"c","08/03/07"
4,"d","08/04/07"
5,"e","08/05/07"
6,"f","08/06/07"
7,"g","08/07/07"
8,"h","08/08/07"
9,"i","08/09/07"
10,"j","08/10/07"
```

Quoting

There are four different quoting options, defined as constants in the csv module.

QUOTE_ALL

> Quote everything, regardless of type.

QUOTE_MINIMAL

> Quote fields with special characters (anything that would confuse a parser configured with the same dialect and options). This is the default

QUOTE_NONNUMERIC

> Quote all fields that are not integers or floats. When used with the reader, input fields that are not quoted are converted to floats.

QUOTE_NONE

Do not quote anything on output. When used with the reader, quote characters are included in the field values (normally, they are treated as delimiters and stripped).

Dialects

There is no well-defined standard for comma-separated value files, so the parser needs to be flexible. This flexibility means there are many parameters to control how `csv` parses or writes data. Rather than passing each of these parameters to the reader and writer separately, they are grouped together conveniently into a *dialect* object.

Dialect classes can be registered by name, so that callers of the csv module do not need to know the parameter settings in advance. The complete list of registered dialects can be retrieved with **`list_dialects()`**.

```
import csv

print csv.list_dialects()
```

The standard library includes two dialects: `excel`, and `excel-tabs`. The `excel` dialect is for working with data in the default export format for Microsoft Excel, and also works with OpenOffice or NeoOffice.

```
$ python csv_list_dialects.py

['excel-tab', 'excel']
```

Creating a Dialect

Suppose instead of using commas to delimit fields, the input file uses `|`, like this:

```
"Title 1"|"Title 2"|"Title 3"
1|"first line
second line"|08/18/07
```

A new dialect can be registered using the appropriate delimiter:

```
import csv

csv.register_dialect('pipes', delimiter='|')

with open('testdata.pipes', 'r') as f:
    reader = csv.reader(f, dialect='pipes')
    for row in reader:
        print row
```

and the file can be read just as with the comma-delimited file:

```
$ python csv_dialect.py

['Title 1', 'Title 2', 'Title 3']
['1', 'first line\nsecond line', '08/18/07']
```

Dialect Parameters

A dialect specifies all of the tokens used when parsing or writing a data file. Every aspect of the file format can be specified, from the way columns are delimited to the character used to escape a token.

Attribute	Default	Meaning
delimiter	`,`	Field separator (one character)
doublequote	True	Flag controlling whether quotechar instances are doubled
escapechar	None	Character used to indicate an escape sequence
lineterminator	`\r\n`	String used by writer to terminate a line
quotechar	`"`	String to surround fields containing special values (one character)
quoting	**`QUOTE_MINIMAL`**	Controls quoting behavior described above
skipinitialspace	False	Ignore whitespace after the field

Attribute	Default	Meaning
delimiter		

```
import csv
import sys

csv.register_dialect('escaped', escapechar='\\', doublequote=F
csv.register_dialect('singlequote', quotechar="'", quoting=csv

quoting_modes = dict( (getattr(csv,n), n) for n in dir(csv) if

for name in sorted(csv.list_dialects()):
    print '\nDialect: "%s"\n' % name
    dialect = csv.get_dialect(name)

    print '  delimiter   = %-6r    skipinitialspace = %r' % (d
                                                            d
    print '  doublequote = %-6r    quoting          = %s' % (d
                                                            q
    print '  quotechar   = %-6r    lineterminator   = %r' % (d
                                                            d
    print '  escapechar  = %-6r' % dialect.escapechar
    print

    writer = csv.writer(sys.stdout, dialect=dialect)
    for i in xrange(3):
        writer.writerow(
            ('col1', i, '10/%02d/2010' % i,
             'Contains special chars: " \' %s to be parsed' %
            )
    print
```

This program shows how the same data appears in several different dialects.

```
$ python csv_dialect_variations.py


Dialect: "escaped"

  delimiter   = ','       skipinitialspace = 0
  doublequote = 0         quoting          = QUOTE_NONE
  quotechar   = '"'       lineterminator   = '\r\n'
  escapechar  = '\\'

col1,0,10/00/2010,Contains special chars: \" ' \, to be parsed
col1,1,10/01/2010,Contains special chars: \" ' \, to be parsed
col1,2,10/02/2010,Contains special chars: \" ' \, to be parsed


Dialect: "excel"

  delimiter   = ','       skipinitialspace = 0
  doublequote = 1         quoting          = QUOTE_MINIMAL
  quotechar   = '"'       lineterminator   = '\r\n'
  escapechar  = None

col1,0,10/00/2010,"Contains special chars: "" ' , to be parsed
col1,1,10/01/2010,"Contains special chars: "" ' , to be parsed
col1,2,10/02/2010,"Contains special chars: "" ' , to be parsed


Dialect: "excel-tab"

  delimiter   = '\t'      skipinitialspace = 0
  doublequote = 1         quoting          = QUOTE_MINIMAL
  quotechar   = '"'       lineterminator   = '\r\n'
  escapechar  = None

col1    0       10/00/2010      "Contains special chars: "" '
col1    1       10/01/2010      "Contains special chars: "" '
col1    2       10/02/2010      "Contains special chars: "" '


Dialect: "singlequote"

  delimiter   = ','       skipinitialspace = 0
  doublequote = 1         quoting          = QUOTE_ALL
  quotechar   = "'"       lineterminator   = '\r\n'
  escapechar  = None

'col1','0','10/00/2010','Contains special chars: " '' , to be
'col1','1','10/01/2010','Contains special chars: " '' , to be
'col1','2','10/02/2010','Contains special chars: " '' , to be
```

Automatically Detecting Dialects

The best way to configure a dialect for parsing an input file is to know the

right settings in advance. For data where the dialect parameters are unknown, the **Sniffer** class can be used to make an educated guess. The **sniff()** method takes a sample of the input data and an optional argument giving the possible delimiter characters.

```
import csv
from StringIO import StringIO
import textwrap

csv.register_dialect('escaped', escapechar='\\', doublequote=F
csv.register_dialect('singlequote', quotechar="'", quoting=csv

# Generate sample data for all known dialects

samples = []

for name in sorted(csv.list_dialects()):
    buffer = StringIO()
    dialect = csv.get_dialect(name)
    writer = csv.writer(buffer, dialect=dialect)
    for i in xrange(3):
        writer.writerow(
            ('col1', i, '10/%02d/2010' % i,
             'Contains special chars: " \' %s to be parsed' %
            )
    samples.append( (name, dialect, buffer.getvalue()) )

# Guess the dialect for a given sample, then use the results t
# the data.

sniffer = csv.Sniffer()

for name, expected, sample in samples:
    print '\nDialect: "%s"\n' % name

    dialect = sniffer.sniff(sample, delimiters=',\t')

    reader = csv.reader(StringIO(sample), dialect=dialect)
    for row in reader:
        print row
```

sniff() returns a **Dialect** instance with the settings to be used for parsing the data. The results are not always perfect, as demonstrated by the "escaped" dialect in the example.

```
$ python csv_dialect_sniffer.py


Dialect: "escaped"

['col1', '0', '10/00/2010', 'Contains special chars: \\" \' \\
['col1', '1', '10/01/2010', 'Contains special chars: \\" \' \\
['col1', '2', '10/02/2010', 'Contains special chars: \\" \' \\

Dialect: "excel"

['col1', '0', '10/00/2010', 'Contains special chars: " \' , to
['col1', '1', '10/01/2010', 'Contains special chars: " \' , to
['col1', '2', '10/02/2010', 'Contains special chars: " \' , to

Dialect: "excel-tab"

['col1', '0', '10/00/2010', 'Contains special chars: " \' \t t
['col1', '1', '10/01/2010', 'Contains special chars: " \' \t t
['col1', '2', '10/02/2010', 'Contains special chars: " \' \t t

Dialect: "singlequote"

['col1', '0', '10/00/2010', 'Contains special chars: " \' , to
['col1', '1', '10/01/2010', 'Contains special chars: " \' , to
['col1', '2', '10/02/2010', 'Contains special chars: " \' , to
```

Using Field Names

In addition to working with sequences of data, the **csv** module includes classes for working with rows as dictionaries so that the fields can be named. The **DictReader** and **DictWriter** classes translate rows to dictionaries instead of lists. Keys for the dictionary can be passed in, or inferred from the first row in the input (when the row contains headers).

```
import csv
import sys

f = open(sys.argv[1], 'rt')
try:
    reader = csv.DictReader(f)
    for row in reader:
        print row
finally:
    f.close()
```

The dictionary-based reader and writer are implemented as wrappers around the sequence-based classes, and use the same methods and arguments. The only difference in the reader API is that rows are returned as dictionaries instead of lists or tuples.

```
$ python csv_dictreader.py testdata.csv

{'Title 1': '1', 'Title 3': '08/18/07', 'Title 2': 'a'}
{'Title 1': '2', 'Title 3': '08/19/07', 'Title 2': 'b'}
{'Title 1': '3', 'Title 3': '08/20/07', 'Title 2': 'c'}
{'Title 1': '4', 'Title 3': '08/21/07', 'Title 2': 'd'}
{'Title 1': '5', 'Title 3': '08/22/07', 'Title 2': 'e'}
{'Title 1': '6', 'Title 3': '08/23/07', 'Title 2': 'f'}
{'Title 1': '7', 'Title 3': '08/24/07', 'Title 2': 'g'}
{'Title 1': '8', 'Title 3': '08/25/07', 'Title 2': 'h'}
{'Title 1': '9', 'Title 3': '08/26/07', 'Title 2': 'i'}
```

The **`DictWriter`** must be given a list of field names so it knows how to order the columns in the output.

```
import csv
import sys

f = open(sys.argv[1], 'wt')
try:
    fieldnames = ('Title 1', 'Title 2', 'Title 3')
    writer = csv.DictWriter(f, fieldnames=fieldnames)
    headers = dict( (n,n) for n in fieldnames )
    writer.writerow(headers)
    for i in range(10):
        writer.writerow({ 'Title 1':i+1,
                          'Title 2':chr(ord('a') + i),
                          'Title 3':'08/%02d/07' % (i+1),
                          })
finally:
    f.close()

print open(sys.argv[1], 'rt').read()
```

```
$ python csv_dictwriter.py testout.csv

Title 1,Title 2,Title 3
1,a,08/01/07
2,b,08/02/07
3,c,08/03/07
4,d,08/04/07
5,e,08/05/07
6,f,08/06/07
7,g,08/07/07
8,h,08/08/07
9,i,08/09/07
10,j,08/10/07
```

See also:

csv

The standard library documentation for this module.

PEP 305

CSV File API