csv — Comma-separated Value Files

Purpose: Read and write comma separated value files.

The csv module can be used to work 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.

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

```
# csv_reader.py

import csv
import sys

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

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

```
"Title 1","Title 2","Title 3","Title 4"
1,"a",08/18/07,"å"
2,"b",08/19/07,"∫"
3,"c",08/20/07,"ç"
```

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

```
$ python3 csv_reader.py testdata.csv

['Title 1', 'Title 2', 'Title 3', 'Title 4']
['1', 'a', '08/18/07', 'å']
['2', 'b', '08/19/07', '∫']
['3', 'c', '08/20/07', 'ç']
```

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
```

Fields with line breaks in the input retain the internal line breaks when they are returned by the parser.

```
$ python3 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.

```
# csv_writer.py
```

```
import csv
import sys

unicode_chars = 'åʃç'

with open(sys.argv[1], 'wt') as f:
    writer = csv.writer(f)
    writer.writerow(('Title 1', 'Title 2', 'Title 3', 'Title 4'))
    for i in range(3):
        row = (
            i + 1,
            chr(ord('a') + i),
            '08/{:02d}/07'.format(i + 1),
            unicode_chars[i],
        )
        writer.writerow(row)

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

The output does not look exactly like the exported data used in the reader example because it lacks quotes around some of the values.

```
$ python3 csv_writer.py testout.csv

Title 1,Title 2,Title 3,Title 4
1,a,08/01/07,å
2,b,08/02/07,ʃ
3,c,08/03/07,ç
```

Quoting

The default quoting behavior is different for the writer, so the second and third columns in the previous example are not quoted. To add quoting, set the `quoting` argument to one of the other quoting modes.

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

In this case, `QUOTE_NONNUMERIC` adds quotes around all columns that contain values that are not numbers.

```
$ python3 csv_writer_quoted.py testout_quoted.csv

"Title 1","Title 2","Title 3","Title 4"
1,"a","08/01/07","å"
2,"b","08/02/07","ʃ"
3,"c","08/03/07","ç"
```

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 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()`.

```
# csv_list_dialects.py

import csv

print(csv.list_dialects())
```

The standard library includes three dialects: `excel`, `excel-tabs`, and `unix`. The `excel` dialect is for working with data in the default export format for Microsoft Excel, and also works with LibreOffice. The `unix` dialect quotes all fields with double-quotes and uses `\n` as the record separator.

```
$ python3 csv_list_dialects.py

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

Creating a Dialect

If, instead of using commas to delimit fields, the input file uses pipes (`|`), 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.

```
# csv_dialect.py

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)
```

Using the “pipes” dialect, the file can be read just as with the comma-delimited file.

```
$ python3 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. the table below lists the aspects of the file format that can be specified, from the way columns are delimited to the character used to escape a token.

CSV Dialect Parameters

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 earlier
skipinitialspace	False	Ignore whitespace after the field delimiter

```
# csv_dialect_variations.py

import csv
import sys

csv.register_dialect('escaped',
```

```
csv.register_dialect('escaped',
                     escapechar='\\',
                     doublequote=False,
                     quoting=csv.QUOTE_NONE,
                     )
csv.register_dialect('singlequote',
                     quotechar="'",
                     quoting=csv.QUOTE_ALL,
                     )

quoting_modes = {
    getattr(csv, n): n
    for n in dir(csv)
    if n.startswith('QUOTE_')
}

TEMPLATE = '''\
Dialect: "{name}"

  delimiter   = {dl!r:<6}    skipinitialspace = {si!r}
  doublequote = {dq!r:<6}    quoting          = {qu}
  quotechar   = {qc!r:<6}    lineterminator   = {lt!r}
  escapechar  = {ec!r:<6}
'''

for name in sorted(csv.list_dialects()):
    dialect = csv.get_dialect(name)

    print(TEMPLATE.format(
        name=name,
        dl=dialect.delimiter,
        si=dialect.skipinitialspace,
        dq=dialect.doublequote,
        qu=quoting_modes[dialect.quoting],
        qc=dialect.quotechar,
        lt=dialect.lineterminator,
        ec=dialect.escapechar,
    ))

    writer = csv.writer(sys.stdout, dialect=dialect)
    writer.writerow(
        ('col1', 1, '10/01/2010',
         'Special chars: " \' {} to parse'.format(
             dialect.delimiter))
    )
    print()
```

This program shows how the same data appears when formatted using several different dialects.

```
$ python3 csv_dialect_variations.py

Dialect: "escaped"

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

col1,1,10/01/2010,Special chars: \" ' \, to parse

Dialect: "excel"

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

col1,1,10/01/2010,"Special chars: "" ' , to parse"

Dialect: "excel-tab"

  delimiter   = '\t'      skipinitialspace = 0
```

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

col1    1       10/01/2010      "Special chars: "" '     to parse"

Dialect: "singlequote"

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

'col1','1','10/01/2010','Special chars: " '' , to parse'

Dialect: "unix"

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

"col1","1","10/01/2010","Special chars: "" ' , to parse"
```

Automatically Detecting Dialects

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

```
# csv_dialect_sniffer.py
```

```
import csv
from io import StringIO
import textwrap

csv.register_dialect('escaped',
                     escapechar='\\',
                     doublequote=False,
                     quoting=csv.QUOTE_NONE)
csv.register_dialect('singlequote',
                     quotechar="'",
                     quoting=csv.QUOTE_ALL)

# 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)
    writer.writerow(
        ('col1', 1, '10/01/2010',
         'Special chars " \' {} to parse'.format(
             dialect.delimiter))
    )
    samples.append((name, dialect, buffer.getvalue()))

# Guess the dialect for a given sample, and then use the results
# to parse the data.
sniffer = csv.Sniffer()
for name, expected, sample in samples:
    print('Dialect: "{}"'.format(name))
    print('In: {}'.format(sample.rstrip()))
    dialect = sniffer.sniff(sample, delimiters=',\t')
    reader = csv.reader(StringIO(sample), dialect=dialect)
    print('Parsed:\n  {}\n'.format(
          '\n  '.join(repr(r) for r in next(reader))))
```

`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.

demonstrated by the "escaped" dialect in the example.

```
$ python3 csv_dialect_sniffer.py

Dialect: "escaped"
In: col1,1,10/01/2010,Special chars \" ' \, to parse
Parsed:
  'col1'
  '1'
  '10/01/2010'
  'Special chars \\" \' \\'
  ' to parse'

Dialect: "excel"
In: col1,1,10/01/2010,"Special chars "" ' , to parse"
Parsed:
  'col1'
  '1'
  '10/01/2010'
  'Special chars " \' , to parse'

Dialect: "excel-tab"
In: col1        1       10/01/2010      "Special chars "" '      to parse"
Parsed:
  'col1'
  '1'
  '10/01/2010'
  'Special chars " \' \t to parse'

Dialect: "singlequote"
In: 'col1','1','10/01/2010','Special chars " '' , to parse'
Parsed:
  'col1'
  '1'
  '10/01/2010'
  'Special chars " \' , to parse'

Dialect: "unix"
In: "col1","1","10/01/2010","Special chars "" ' , to parse"
Parsed:
  'col1'
  '1'
  '10/01/2010'
  'Special chars " \' , to parse'
```

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).

```
# csv_dictreader.py

import csv
import sys

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

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 OrderedDict instances instead of lists or tuples (under earlier verison of Python, the rows were returned as regular `dict` instances).

```
$ python3 csv_dictreader.py testdata.csv

OrderedDict([('Title 1', '1'), ('Title 2', 'a'), ('Title 3',
'08/18/07'), ('Title 4', 'å')])
OrderedDict([('Title 1', '2'), ('Title 2', 'b'), ('Title 3',
```

```
'08/19/07'), ('Title 4', 'ʃ')])
OrderedDict([('Title 1', '3'), ('Title 2', 'c'), ('Title 3',
'08/20/07'), ('Title 4', 'ç')])
```

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

```
# csv_dictwriter.py

import csv
import sys

fieldnames = ('Title 1', 'Title 2', 'Title 3', 'Title 4')
headers = {
    n: n
    for n in fieldnames
}
unicode_chars = 'åʃç'

with open(sys.argv[1], 'wt') as f:

    writer = csv.DictWriter(f, fieldnames=fieldnames)
    writer.writeheader()

    for i in range(3):
        writer.writerow({
            'Title 1': i + 1,
            'Title 2': chr(ord('a') + i),
            'Title 3': '08/{:02d}/07'.format(i + 1),
            'Title 4': unicode_chars[i],
        })

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

The field names are not written to the file automatically, but they can be written explicitly using the `writeheader()` method.

```
$ python3 csv_dictwriter.py testout.csv

Title 1,Title 2,Title 3,Title 4
1,a,08/01/07,å
2,b,08/02/07,ʃ
3,c,08/03/07,ç
```

See also

- Standard library documentation for csv
- **PEP 305** – CSV File API
- Python 2 to 3 porting notes for csv

Building Documents With Element Nodes

Data Compression and Archiving

This page was last updated 2018-03-18.

Navigation
Building Documents With Element Nodes
Data Compression and Archiving

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?