Importing CSV Data into Python Using csv & pandas

Xavier Morera
BIG DATA INC.

@xmorera www.xaviermorera.com

```
import csv

dir(csv)
```

Python csv API

Implements classes to read (and write) tabular data in CSV format

- PEP 305 - CSV File API

PEP 305 -- CSV File API

PEP:	305
Title:	CSV File API
Author:	Kevin Altis <altis at semi-retired.com>, Dave Cole <djc at object-craft.com.au>, Andrew McNamara <andrewm at object-craft.com.au>, Skip Montanaro <skip at pobox.com>, Cliff Wells <LogiplexSoftware at earthlink.net>
Discussions-To:	<csv at python.org>
Status:	Final
Type:	Standards Track
Created:	26-Jan-2003

Application Domain

This PEP is about doing one thing well: parsing tabular data which may use a variety of field separators, quoting characters, quote escape mechanisms and line endings. The authors intend the proposed module to solve this one parsing problem efficiently. The authors do not intend to address any of these related topics:

- data interpretation (is a field containing the string "10" supposed to be a string, a float or an int? is it a number in base 10, base 16 or base 2? is a number in quotes a number or a string?)
- locale-specific data representation (should the number 1.23 be written as "1.23" or "1,23" or "1 23"?) -- this may eventually be addressed.
- fixed width tabular data - can already be parsed reliably.

Rationale

Often, CSV files are formatted simply enough that you can get by reading them line-by-line and splitting on the commas which delimit the fields. This is especially true if all the data being read is numeric. This approach may work for a while, then come back to bite you in the butt when somebody puts something unexpected in the data like a comma. As you dig into the problem you may eventually come to the conclusion that you can solve the problem using regular expressions. This will work for a while, then break mysteriously one day. The problem grows, so you dig deeper and eventually realize that you need a purpose-built parser for the format.

CSV formats are not well-defined and different implementations have a number of subtle corner cases. It has been suggested that the "V" in the acronym stands for "Vague" instead of "Values". Different delimiters and quoting characters are just the start. Some programs generate whitespace after each delimiter which is not part of the following field. Others quote embedded quoting characters by doubling them, others by prefixing them with an escape character. The list of weird ways to do things can seem endless.

All this variability means it is difficult for programmers to reliably parse CSV files from many sources or generate CSV files designed to be fed to specific external programs without a thorough understanding of those sources and programs. This PEP

Reading CSV Files

CSV readers are created with the reader factory function:

```
obj = reader(iterable [, dialect='excel']
             [optional keyword args])
```

A reader object is an iterator which takes an iterable object returning lines as the sole required parameter. If it supports a binary mode (file objects do), the iterable argument to the reader function must have been opened in binary mode. This gives the reader object full control over the interpretation of the file's contents. The optional dialect parameter is discussed below. The reader function also accepts several optional keyword arguments which define specific format settings for the parser (see the section "Formatting Parameters"). Readers are typically used as follows:

```
csvreader = csv.reader(file("some.csv"))
for row in csvreader:
    process(row)
```

Each row returned by a reader object is a list of strings or Unicode objects.

When both a dialect parameter and individual formatting parameters are passed to the constructor, first the dialect is queried for formatting parameters, then individual formatting parameters are examined.

Writing CSV Files

Creating writers is similar:

```
obj = writer(fileobj [, dialect='excel'],
```

```
import csv

dir(csv)
```

Python csv API

Implements classes to read (and write) tabular data in CSV format

- PEP 305 - CSV File API

CSV Reader used for loading data, with small differences

- For example loading a CSV generated by Excel (dialects)

```
54
55      class excel(Dialect):
56          """Describe the usual properties of Excel-generated CSV files."""
57          delimiter = ','
58          quotechar = '"'
59          doublequote = True
60          skipinitialspace = False
61          lineterminator = '\r\n'
62          quoting = QUOTE_MINIMAL
63      register_dialect("excel", excel)
64
65      class excel_tab(excel):
66          """Describe the usual properties of Excel-generated TAB-delimited files."""
67          delimiter = '\t'
68      register_dialect("excel-tab", excel_tab)
69
70      class unix_dialect(Dialect):
71          """Describe the usual properties of Unix-generated CSV files."""
72          delimiter = ','
73          quotechar = '"'
74          doublequote = True
75          skipinitialspace = False
76          lineterminator = '\n'
77          quoting = QUOTE_ALL
```

```
csv.list_dialects()
with open('users-simple-five.csv') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',', quotechar='"')
    for row in csv_reader:
        print(row)

with open('users-simple-five.csv') as csv_file:
    csv_reader = csv.reader(csv_file, dialect='excel')
    for row in csv_reader:
        print(row)

field_names = ['Id', 'Reputation', 'Location', 'DisplayName']
with open('users-simple-five.csv') as csv_file:
    csv_dict_reader = csv.DictReader(csv_file, fieldnames=field_names)
    for row in csv_dict_reader:
        print(row['DisplayName'] + ' has a reputation of ' + row['Reputation'])
```

Importing CSV Files Using the csv Module

Load using reader()

- Returns a reader object that can be used to iterate over the lines

Can also load as a DictReader()

- Specify fieldnames

```
with open('users-simple-five.csv') as csv_file:
    csv_reader = csv.reader(csv_file, quoting=csv.QUOTE_NONNUMERIC)
    for row in csv_reader:
        print(row)

with open('users-five.csv') as csv_file:
    csv_reader = csv.reader(csv_file, quoting=csv.QUOTE_NONNUMERIC)
    for row in csv_reader:
        print(row)
```

Importing CSV Files Using the csv Module

Values returned as strings

- You can load unquoted as numbers
- But you can run into issues
- Depending on the quoting used when writing the CSV

Table Of Contents

csv — CSV File Reading and Writing

- Module Contents
- Dialects and Formatting Parameters
- Reader Objects
- Writer Objects
- Examples

Previous topic

File Formats

Next topic

configparser — Configuration file parser

csv — CSV File Reading and Writing

Source code: Lib/csv.py

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 API
The Python Enhancement Proposal which proposed this addition to Python.

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

Table Of Contents

csv — CSV File Reading and Writing
- Module Contents
- Dialects and Formatting Parameters
- Reader Objects
- Writer Objects
- Examples

Previous topic
File Formats

Next topic
configparser — Configuration file parser

This Page

Report a Bug
Show Source

csv — CSV File Reading and Writing

Source code: Lib/csv.py

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 API
The Python Enhancement Proposal which proposed this addition to Python.

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

Python's csv module is good

Useful, easy to use...

It has the functionality you need

Use a reader or DictReader

But there are a few drawbacks

What other libraries are available?

Importing Text & CSV Files

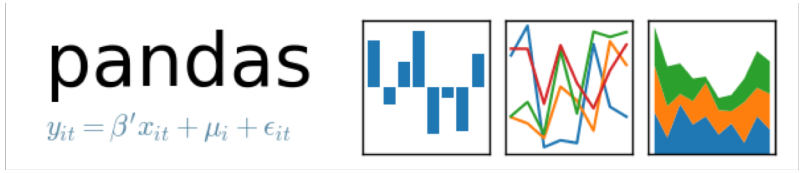
So far...

- Import data with Numpy into ndarray
- Import data with csv using a reader

Both have a very specific purpose

- But usually we need a little bit more

Importing Text & CSV Files Using Pandas



Tabular data

- Relational or labeled data
- Columns of different types

Perform operations on your data

- Data manipulation and analysis
- Numerical and time-series data

DataFrame is the first class citizen

Pandas logo: https://github.com/pandas-dev/pandas/commit/ab4ac12

import pandas as pd

Importing Text & CSV Files Using Pandas

Load CSV data using read_csv()

- Many parameters available to specify from where, number of rows and columns, which rows, which columns, what types, and more

```
posts_csv = pd.read_csv('posts-100.csv')

type(posts_csv)

posts_csv

posts_csv.head()

posts_csv.head(2)
```

Importing with read_csv()

Specify the filename

Loaded into a DataFrame

All rows loaded, but you can specify subset with head()

```
posts_csv.values()

type(posts_csv.values())
```

The ndarray in Your pandas DataFrame

DataFrame depends and interoperates with NumPy

- Return a NumPy representation of the DataFrame
- With values, but axes labels are removed

```
remote_file = 'https://raw.githubusercontent.com/xmorera/sample-data/master/csv/posts-100.csv'

posts_url = pd.read_csv(remote_file, header=None)

posts_url.head()
```

Load from URL

Load data from a remote file using a URL

Valid URL schemes

- http, ftp, s3, and file

```
posts_small = pd.read_csv('posts-100.csv', nrows=3)

posts_small

posts_small = pd.read_csv('posts-100.csv', nrows=3, skiprows=3)

posts_small
```

Read Pieces of Large Files

Specify how many rows to read, with nrows

And where to start, with skiprows

Can also specify skipfooter to ignore lines at the end

```
posts_odd = pd.read_csv('posts-100.csv', skiprows=lambda x: x % 2 != 0)

posts_odd.head()
```

Specify Rows Using a Function

More advanced method for specifying which rows to load

Referred as callable

- Use a named function or an anonymous function
- Evaluate against row indices and determine which rows to skip

```
posts_columns = pd.read_csv('posts-100.csv', usecols=[0,6,7,8])

posts_columns.head(5)

posts_columns.columns
```

Loading (Certain) Columns

Specify which columns to load

- With usecols
- List or a function

Columns get a name (but something does not look right)

```
posts_no_header = pd.read_csv('posts-100.csv', header=None)
posts_no_header.columns
posts_prefix = pd.read_csv('posts-100.csv', header=None, prefix='Col’)
posts_prefix.columns
header_fields = ['New_Id', 'New_PostTypeId', 'New_CreationDate', 'New_Score', 'New_ViewCount',
'New_LastActivityDate', 'New_Title', 'New_Tags', 'New_AnswerCount', 'New_CommentCount',
'New_FavoriteCount', 'New_ClosedDate']
posts_add_header = pd.read_csv('posts-100.csv', names=header_fields)
posts_add_header
```

Column Headers Not Included in File

Specify that file does not include column header

- One automatically assigned
- Set a prefix

Provide column names

```
# head posts-100-header.csv

posts_header = pd.read_csv('posts-100-header.csv’)

posts_header.columns

posts_header[['Id', 'AnswerCount']].head()

pd.read_csv('posts-100-header.csv', header='infer').columns
```

Headers Included in File

Some files include headers

- Use them, they are useful for referring to columns

Can infer header

```
pd.read_csv('posts-100-header.csv', usecols=[0, 1, 2, 7]).dtypes

pd.read_csv('posts-100-header.csv', usecols=[0, 1, 2, 7], dtype={'PostTypeId': str}).dtypes

pd.read_csv('posts-100-header.csv', usecols=[0, 1, 2, 7], dtype={'PostTypeId': float}).dtypes
```

Specify Column Types on Load

Types inferred on load

- **Review using dtypes**

You can manually set specific types using dtype

```
posts_tags = pd.read_csv('posts-100-header.csv', usecols=[0, 1, 2, 7])
posts_tags.iloc[1]
posts_tags.iloc[1]['Tags']
type(posts_tags.iloc[1]['Tags'])
import re
posts_tags = pd.read_csv('posts-100-header.csv', usecols=[0, 1, 2, 7], converters={'Tags': lambda x:
re.findall('<[A-Za-z0-9_-]*>',x)})
type(posts_tags .iloc[1]['Tags'])
posts_tags .iloc[1]['Tags']
```

Apply Function to a Column

Take a string and convert to a list

Use converters to apply functions on columns

Very powerful!

```
posts_date = pd.read_csv('posts-100-header.csv', usecols=[0, 1, 2, 7])

posts_date

type(posts_date['CreationDate'][0])

posts_date = pd.read_csv('posts-100-header.csv', usecols=[0, 1, 2, 7],
parse_dates=['CreationDate'])

posts_date.dtypes

type(posts_date['CreationDate'][0])
```

Loading Dates

Dates and times are important, yet they can be complex to deal with

- May be imported as strings

Use parse_dates

- Index(es) or name(s)

```
pd.read_csv('posts-100-header.csv', usecols=[0, 3, 4, 8, 9, 10]).head(5)

pd.read_csv('posts-100-header.csv', usecols=[0, 3, 4, 8, 9, 10], na_filter=False).head()

pd.read_csv('posts-100-header.csv', usecols=[0, 3, 4, 8, 9, 10], na_filter=True).head()

posts_missing = pd.read_csv('posts-100-header.csv', usecols=[0, 3, 4, 8, 9, 10], dtype={'ViewCount':float},
na_filter=False)

posts_missing = pd.read_csv('posts-100-header.csv', usecols=[0, 3, 4, 8, 9, 10], dtype={'ViewCount':float},
na_filter=True)
```

Missing Values

Data is not perfect, there may be missing values: NaN or empty

Detect missing value markers with na_filter

Use na_values for other markers and keep_default_na to include default values

```
# head –n2 'posts-100.tsv
pd.read_csv('posts-100.tsv').head()
pd.read_csv('posts-100.tsv', sep='\t').head()
pd.read_csv('posts-100.tsv', delimiter='\t').head()
posts_tsv = pd.read_table('posts-100.tsv')
posts_tsv.head()
```

Tabular Data

Other delimiters available

- Set \t for tab, with sep or delimiter

Use read_table for tab separated values

Table Of Contents

- What's New
- Installation
- Contributing to pandas
- Package overview
- 10 Minutes to pandas
- Tutorials
- Cookbook
- Intro to Data Structures
- Essential Basic Functionality
- Working with Text Data
- Options and Settings
- Indexing and Selecting Data
- MultiIndex / Advanced Indexing
- Computational tools
- Working with missing data
- Group By: split-apply-combine
- Merge, join, and concatenate
- Reshaping and Pivot Tables
- Time Series / Date functionality
- Time Deltas
- Categorical Data
- Visualization
- Styling
- IO Tools (Text, CSV, HDF5, …)
- Enhancing Performance
- Sparse data structures
- Frequently Asked Questions (FAQ)
- rpy2 / R interface
- pandas Ecosystem
- Comparison with R / R libraries
- Comparison with SQL
- Comparison with SAS
- Comparison with Stata
- API Reference
  - Input/Output
    - Pickling
    - Flat File
      - pandas.read_table

pandas.read_csv

`pandas.`**`read_csv`**(*filepath_or_buffer*, *sep=', '*, *delimiter=None*, *header='infer'*, *names=None*, *index_col=None*, *usecols=None*, *squeeze=False*, *prefix=None*, *mangle_dupe_cols=True*, *dtype=None*, *engine=None*, *converters=None*, *true_values=None*, *false_values=None*, *skipinitialspace=False*, *skiprows=None*, *nrows=None*, *na_values=None*, *keep_default_na=True*, *na_filter=True*, *verbose=False*, *skip_blank_lines=True*, *parse_dates=False*, *infer_datetime_format=False*, *keep_date_col=False*, *date_parser=None*, *dayfirst=False*, *iterator=False*, *chunksize=None*, *compression='infer'*, *thousands=None*, *decimal=b'.'*, *lineterminator=None*, *quotechar='"'*, *quoting=0*, *escapechar=None*, *comment=None*, *encoding=None*, *dialect=None*, *tupleize_cols=None*, *error_bad_lines=True*, *warn_bad_lines=True*, *skipfooter=0*, *doublequote=True*, *delim_whitespace=False*, *low_memory=True*, *memory_map=False*, *float_precision=None*) [source]

Read CSV (comma-separated) file into DataFrame

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for IO Tools.

filepath_or_buffer : *str, pathlib.Path, py._path.local.LocalPath or any* \

object with a read() method (such as a file handle or StringIO)

The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. For instance, a local file could be file://localhost/path/to/table.csv

sep *: str, default ','*

Delimiter to use. If sep is None, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python's builtin sniffer tool, `csv.Sniffer`. In addition, separators longer than 1 character and different from `'\s+'` will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: `'\r\t'`

delimiter *: str, default* `None`

Alternative argument name for sep.

delim_whitespace *: boolean, default False*

Specifies whether or not whitespace (e.g. `' '` or `'\t'`) will be used as the sep.