Lesen und Schreiben von CSV-Dateien

Quellcode: Lib/csv.py

https://realpython.com/python-csv/

Das sogenannte CSV-Format (Comma Separated Values) ist das gebräuchlichste Import- und Exportformat für Tabellenkalkulationen und Datenbanken. Das CSV-Format wurde für viele verwendet Jahre vor Versuchen, das Format standardisiert zu beschreiben **RFC 4180** . Das Fehlen eines klar definierten Standards bedeutet, dass subtile Unterschiede bestehen sind oft in den Daten vorhanden, die von verschiedenen Anwendungen produziert und verarbeitet werden. Diese Unterschiede können es lästig machen, CSV-Dateien aus mehreren Quellen zu verarbeiten. Während die Trennzeichen und Anführungszeichen variieren, ist das Gesamtformat jedoch unterschiedlich ähnlich genug, dass es möglich ist, ein einzelnes Modul zu schreiben, das kann manipulieren Sie solche Daten effizient und verbergen Sie die Details des Lesens und Schreibens der Daten vom Programmiergerät.

Das `csv`-Modul implementiert Klassen zum Lesen und Schreiben von Tabellendaten in CSV Format. Programmierer können sagen: „Schreiben Sie diese Daten im bevorzugten Format von Excel“ oder „Daten aus dieser Datei lesen, die von Excel generiert wurden“ ohne Kenntnis der genauen Details des von Excel verwendeten CSV-Formats. Programmierer können beschreiben auch die von anderen Anwendungen verstandenen CSV-Formate oder definieren deren eigene spezielle CSV-Formate.

Das `csv`Moduls `reader`und `writer`Objekte lesen und Sequenzen schreiben. Programmierer können auch Daten in Wörterbuchform lesen und schreiben Verwendung der `DictReader`und `DictWriter`Klassen.

Siehe auch

PEP 305 - CSV-Datei-API

> Der Python-Verbesserungsvorschlag, der diese Ergänzung zu Python vorschlug.

Modulinhalte

Das `csv`Modul definiert die folgenden Funktionen:

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

> ein Reader-Objekt zurück, das Zeilen in der angegebenen *CSV-Datei durchläuft* . *csvfile* kann jedes Objekt sein, das das Iteratorprotokoll und eine zurückgibt String jedes Mal seine `__next__()`-Methode aufgerufen wird — Dateiobjekte als auch Listenobjekte sind geeignet. Wenn *csvfile* ein Dateiobjekt ist, es sollte mit geöffnet werden `newline=''`. 1 Optional *Dialektparameter* angegeben werden, der verwendet wird, um einen Satz von

Parametern zu definieren spezifisch für einen bestimmten CSV-Dialekt. Es kann eine Instanz einer Unterklasse von sein das `Dialect`Klasse oder eine der von der zurückgegebenen Zeichenfolgen `list_dialects()`Funktion. Die anderen optionalen *fmtparams*-Schlüsselwortargumente kann gegeben werden, um einzelne Formatierungsparameter in der aktuellen zu überschreiben Dialekt. Ausführliche Informationen zu den Dialekt- und Formatierungsparametern finden Sie unter Abschnitt Dialekte und Formatierungsparameter .

Jede aus der CSV-Datei gelesene Zeile wird als Liste von Zeichenfolgen zurückgegeben. Nein automatische Datentypkonvertierung wird durchgeführt, es sei denn, die `QUOTE_NONNUMERIC`Format Option angegeben ist (in diesem Fall werden Felder ohne Anführungszeichen in Gleitkommazahlen umgewandelt).

Ein kurzes Anwendungsbeispiel:

```
>>>
```

```
>>> 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*)

Geben Sie ein Writer-Objekt zurück, das für die Konvertierung der Benutzerdaten in begrenzte Daten verantwortlich ist Zeichenfolgen für das angegebene dateiähnliche Objekt. *csvfile* kann ein beliebiges Objekt mit einem sein `write()`Methode. Wenn *csvfile* ein Dateiobjekt ist, sollte es mit geöffnet werden `newline=''` 1 . Ein optionaler *Dialekt* Es kann ein Parameter angegeben werden, der verwendet wird, um eine Reihe von Parametern zu definieren, die für a spezifisch sind bestimmten CSV-Dialekt. Es kann eine Instanz einer Unterklasse von sein `Dialect`Klasse oder eine der von der zurückgegebenen Zeichenfolgen `list_dialects()`Funktion. Die anderen optionalen *fmtparams*- Schlüsselwortargumente kann gegeben werden, um einzelne Formatierungsparameter in der aktuellen zu überschreiben Dialekt. Ausführliche Informationen zu Dialekten und Formatierungsparametern finden Sie unter Abschnitt Dialekte und Formatierungsparameter . Um es zu schaffen möglichst einfach mit Modulen zu verbinden, die die DB-API implementieren, die Wert `None`wird als leerer String geschrieben. Dies ist zwar kein umkehrbare Transformation erleichtert es, SQL-NULL-Datenwerte zu sichern CSV-Dateien ohne Vorverarbeitung der von a `cursor.fetch*`Anruf. Alle anderen Nicht-String-Daten werden mit gestringt `str()`bevor geschrieben wird.

Ein kurzes Anwendungsbeispiel:

```
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* [, *Dialekt* [, ***fmtparams*]])

Assoziiere *Dialekt* mit *Namen* . *Name* muss eine Zeichenfolge sein. Das Dialekt kann entweder durch Übergabe einer Unterklasse von angegeben werden `Dialect`, oder durch *fmtparams*- Schlüsselwortargumente oder beides, wobei Schlüsselwortargumente überschrieben werden Parameter des Dialekts. Ausführliche Informationen zu Dialekten und Formatierung Parameter, siehe Abschnitt Dialekte und Formatierungsparameter .

`csv.unregister_dialect`(*Name*)

Löschen Sie den Dialekt, der dem *Namen* aus der Dialektregistrierung. Ein `Error`wird ausgelöst, wenn *name* kein registrierter Dialektname ist.

`csv.get_dialect`(*Name*)

Gibt den mit *name* . Ein `Error`wird erhoben, wenn *name* ist kein registrierter Dialektname. Diese Funktion gibt eine unveränderliche zurück `Dialect`.

`csv.list_dialects`( )

Gibt die Namen aller registrierten Dialekte zurück.

`csv.field_size_limit`([*new_limit*])

Gibt die aktuell vom Parser zugelassene maximale Feldgröße zurück. Wenn *new_limit* ist gegeben, wird dies der neue Grenzwert.

Das `csv`Modul definiert die folgenden Klassen:

Klasse `csv.DictReader`(*f* , *fieldnames = None* , *restkey = None* , *restval = None* , *dialect = 'excel'* , ** args* , *** kwds*)

Erstellen Sie ein Objekt, das wie ein normales Lesegerät funktioniert, aber das abbildet Informationen in jeder Zeile zu a `dict`deren Schlüssel von der gegeben werden optionaler *Feldnamen*- Parameter.

Der *fieldnames*- Parameter ist eine Sequenz . Wenn *Feldnamen* ist weggelassen, werden die Werte in der ersten Zeile der Datei *f* als verwendet Feldnamen. Unabhängig davon, wie die Feldnamen bestimmt werden, die Wörterbuch behält ihre ursprüngliche Reihenfolge bei.

Wenn eine Zeile mehr Felder als Feldnamen hat, werden die restlichen Daten in a eingefügt list und mit dem durch restkey angegebenen *Feldnamen gespeichert* (was zu `None`). Wenn eine nicht leere Zeile weniger Felder als Feldnamen hat, wird die fehlende Werte werden mit dem Wert von *restval aufgefüllt* (was zu `None`).

Alle anderen optionalen oder Schlüsselwort-Argumente werden an den Basiswert übergeben `reader`Beispiel.

Geändert in Version 3.6: Zurückgegebene Zeilen sind jetzt vom Typ `OrderedDict`.

Geändert in Version 3.8: Zurückgegebene Zeilen sind jetzt vom Typ `dict`.

Ein kurzes Anwendungsbeispiel:

```
>>>
```

```
>>> import csv
>>> with open('names.csv', newline='') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Eric Idle
John Cleese

>>> print(row)
{'first_name': 'John', 'last_name': 'Cleese'}
```

Klasse `csv.DictWriter`(*f* , *fieldnames* , *restval = ''* , *extrasaction = 'raise'* , *dialect = 'excel'* , * *args* , ** *kwds*)

Erstellen Sie ein Objekt, das wie ein normaler Writer funktioniert, aber Wörterbücher abbildet auf Ausgabezeilen. Der *fieldnames*- Parameter ist a `sequence`von Schlüsseln, die die Reihenfolge identifizieren, in der Werte in der Wörterbuch übergeben an die `writerow()`Methode werden in die Datei geschrieben *f* . Der optionale *restval*- Parameter gibt den Wert an, der sein soll geschrieben, wenn dem Wörterbuch ein Schlüssel in *Feldnamen fehlt* . Wenn die Wörterbuch übergeben an die `writerow()`Methode enthält einen Schlüssel, der nicht gefunden wurde *fieldnames* , der optionale *extrasaction* gibt an, welche Aktion ausgeführt werden soll nehmen. Wenn es eingestellt ist `'raise'`, der Standardwert, a `ValueError`wird angehoben. Wenn es eingestellt ist `'ignore'`, werden zusätzliche Werte im Wörterbuch ignoriert. Alle anderen optionalen oder Schlüsselwortargumente werden an den zugrunde liegenden Wert übergeben `writer`Beispiel.

Beachten Sie, dass im Gegensatz zu `DictReader`Klasse, der *fieldnames*- Parameter des `DictWriter`Klasse ist nicht optional.

Ein kurzes Anwendungsbeispiel:

```
import csv

with open('names.csv', 'w', newline='') 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'})
```

Klasse `csv.Dialect`

Das `Dialect`Klasse ist eine Containerklasse, deren Attribute enthalten Informationen zum Umgang mit doppelten Anführungszeichen, Leerzeichen, Trennzeichen usw. Aufgrund des Fehlens einer strikten CSV-Spezifikation sind unterschiedliche Anwendungen möglich subtil unterschiedliche CSV-Daten erzeugen. `Dialect`Instanzen definieren wie `reader`und `writer`Instanzen verhalten sich.

Alles Verfügbar `Dialect`Namen werden von zurückgegeben `list_dialects()`, und sie können mit spezifischen registriert werden `reader`und `writer` Klassen durch ihren Initialisierer (`__init__`) funktioniert so:

```
import csv

with open('students.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile, dialect='unix')
                                 ^^^^^^^^^^^^^^
```

Klasse `csv.excel`

Das `excel`-Klasse definiert die üblichen Eigenschaften einer Excel-generierten CSV-Datei Datei. Es wird mit dem Dialektnamen registriert `'excel'`.

Klasse `csv.excel_tab`

Das `excel_tab`Klasse definiert die üblichen Eigenschaften einer Excel-generierten TAB-getrennte Datei. Es wird mit dem Dialektnamen registriert `'excel-tab'`.

Klasse `csv.unix_dialect`

Das `unix_dialect`Die Klasse definiert die üblichen Eigenschaften einer CSV-Datei auf UNIX-Systemen generiert, dh mit `'\n'`als Zeilenabschluss und Anführungszeichen alle Felder. Es wird mit dem Dialektnamen registriert `'unix'`.

Neu in Version 3.2.

Klasse `csv.Sniffer`

Das `Sniffer`-Klasse wird verwendet, um das Format einer CSV-Datei abzuleiten.

Das `Sniffer`Die Klasse bietet zwei Methoden:

`sniff`(*Beispiel* , *Trennzeichen* = *Keine*)

Analysieren Sie die gegebene *Probe* und senden Sie a `Dialect`Unterklasse die gefundenen Parameter widerspiegeln. Wenn der *Trennzeichenparameter* optionale gegeben ist, wird sie als Zeichenkette interpretiert, die mögliche Gültigkeiten enthält Trennzeichen.

`has_header`(*Probe*)

Analysieren Sie den Beispieltext (vermutlich im CSV-Format) und kehren Sie zurück `True`wenn die erste Zeile eine Reihe von Spaltenüberschriften zu sein scheint. Bei der Inspektion jeder Spalte wird eines von zwei Schlüsselkriterien berücksichtigt Schätzen Sie, ob das Beispiel einen Header enthält:

- die zweite bis n-te Zeile enthalten numerische Werte

- die zweite bis n-te Zeile enthalten Zeichenfolgen mit mindestens einem Wert Länge unterscheidet sich von der des mutmaßlichen Headers dieser Spalte.

Zwanzig Reihen nach der ersten Reihe werden abgetastet; wenn mehr als die Hälfte der Spalten + Zeilen erfüllen die Kriterien, `True`ist zurück gekommen.

Notiz

Diese Methode ist eine grobe Heuristik und kann sowohl falsch positive Ergebnisse als auch erzeugen Negative.

Ein Beispiel für `Sniffer`verwenden:

```
with open('example.csv', newline='') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... process CSV file contents here ...
```

Das `csv`Modul definiert die folgenden Konstanten:

`csv.QUOTE_ALL`

Weist an `writer`Objekte, um alle Felder zu zitieren.

`csv.QUOTE_MINIMAL`

Weist an `writer`Objekte, um nur die Felder zu zitieren, die enthalten Sonderzeichen wie *Trennzeichen* , *Anführungszeichen* oder eines der Zeichen in *Zeilenabschluss* .

`csv.QUOTE_NONNUMERIC`

Weist an `writer`Objekte, um alle nicht numerischen Felder in Anführungszeichen zu setzen.

umzuwandeln *float* .

`csv.QUOTE_NONE`

Weist an `writer`Objekte, Felder niemals in Anführungszeichen zu setzen. Wenn der Strom *delimiter* tritt in Ausgabedaten auf, ihm geht das aktuelle *Escapechar voraus* Charakter. Wenn *escapechar* nicht gesetzt ist, löst der Writer aus `Error`wenn Es werden alle Zeichen gefunden, die mit Escapezeichen versehen werden müssen.

Weist an `reader`keine spezielle Verarbeitung von Anführungszeichen durchzuführen.

Das `csv`Modul definiert die folgende Ausnahme:

Ausnahme `csv.Error`

Wird von einer der Funktionen ausgelöst, wenn ein Fehler erkannt wird.

Dialekte und Formatierungsparameter

Um das Festlegen des Formats von Eingabe- und Ausgabedatensätzen zu erleichtern, spezifisch Formatierungsparameter sind in Dialekten zusammengefasst. Ein Dialekt ist ein Unterklasse der `Dialect`Klasse mit einem Satz spezifischer Methoden und a Single `validate()`Methode. Beim Erstellen `reader`oder `writer`Objekte, kann der Programmierer eine Zeichenkette oder eine Unterklasse davon spezifizieren das `Dialect`class als Dialektparameter. Zusätzlich oder stattdessen des *Dialektparameters* kann der Programmierer auch individuell vorgeben Formatierungsparameter, die die gleichen Namen haben wie die unten definierten Attribute für die `Dialect`Klasse.

Dialekte unterstützen die folgenden Attribute:

`Dialect.delimiter`

> Eine aus einem Zeichen bestehende Zeichenfolge, die zum Trennen von Feldern verwendet wird. Es ist standardmäßig auf `','`.

`Dialect.doublequote`

> wie Instanzen von *quotechar* , die in einem Feld erscheinen sollen selbst zitiert werden. Wann `True`, wird das Zeichen verdoppelt. Wann `False`, das *Escapechar* wird als Präfix für das *Quotechar verwendet* . Es standardmäßig auf `True`.
>
> Bei der Ausgabe, wenn *doppelte Anführungszeichen vorhanden* sind `False`und kein *Escapechar* gesetzt ist, `Error`wird ausgelöst, wenn in einem Feld ein *Quotechar* gefunden wird.

`Dialect.escapechar`

> Eine aus einem Zeichen bestehende Zeichenfolge, die vom Schreiber verwendet wird, um das *Trennzeichen* bei *maskieren* ist eingestellt auf `QUOTE_NONE`und das *Anführungszeichen* , wenn *doppelte Anführungszeichen* sind `False`. Beim Lesen entfernt das *Escapechar* jede besondere Bedeutung von das folgende Zeichen. Es ist standardmäßig auf `None`, wodurch das Entkommen deaktiviert wird.

`Dialect.lineterminator`

> Die Zeichenfolge, die zum Beenden von Zeilen verwendet wird, die von der erzeugt werden `writer`. Es ist standardmäßig zu `'\r\n'`.
>
> Notiz
>
> Das `reader`ist fest codiert, um entweder zu erkennen `'\r'`oder `'\n'`wie Zeilenende und ignoriert *lineterminator* . Dieses Verhalten kann sich in ändern Zukunft.

`Dialect.quotechar`

> Eine aus einem Zeichen bestehende Zeichenfolge, die verwendet wird, um Felder zu zitieren, die Sonderzeichen enthalten, wie z als *Trennzeichen* oder *Anführungszeichen* enthalten. Es standardmäßig auf `'"'`.

`Dialect.quoting`

Steuert, wann Zitate vom Schreiber generiert und von der erkannt werden sollen Leser. Es kann jede der aufnehmen `QUOTE_*`Konstanten (siehe Abschnitt Module Contents) und standardmäßig auf `QUOTE_MINIMAL`.

`Dialect.skipinitialspace`

Wann `True`, Leerzeichen unmittelbar nach dem *Trennzeichen* werden ignoriert. Die Voreinstellung ist `False`.

`Dialect.strict`

Wann `True`, Ausnahme auslösen `Error`bei fehlerhafter CSV-Eingabe. Die Voreinstellung ist `False`.

Reader-Objekte

Reader-Objekte (`DictReader`Instanzen und Objekte, die von zurückgegeben werden `reader()`Funktion) haben die folgenden öffentlichen Methoden:

`csvreader.__next__()`

Gibt die nächste Zeile des iterierbaren Objekts des Readers als Liste zurück (wenn das Objekt object kam zurück `reader()`) oder ein Diktat (falls es sich um eine `DictReader` Instanz), nach dem aktuellen geparst `Dialect`. Normalerweise du sollte dies als nennen `next(reader)`.

Reader-Objekte haben die folgenden öffentlichen Attribute:

`csvreader.dialect`

Eine schreibgeschützte Beschreibung des vom Parser verwendeten Dialekts.

`csvreader.line_num`

Die Anzahl der vom Quell-Iterator gelesenen Zeilen. Dies ist nicht dasselbe wie die Anzahl der zurückgegebenen Datensätze, da Datensätze mehrere Zeilen umfassen können.

DictReader-Objekte haben das folgende öffentliche Attribut:

`csvreader.fieldnames`

Wenn dieses Attribut beim Erstellen des Objekts nicht als Parameter übergeben wird initialisiert beim ersten Zugriff oder wenn der erste Datensatz aus dem gelesen wird Datei.

Writer-Objekte

`Writer`Objekte (`DictWriter`Instanzen und Objekte, die von zurückgegeben werden das `writer()`Funktion) haben die folgenden öffentlichen Methoden. Eine *Reihe* muss sein ein

Iterable von Strings oder Zahlen für `Writer`Objekte und ein Wörterbuch Zuordnen von Feldnamen zu Zeichenfolgen oder Zahlen (indem sie durchgeleitet werden `str()` zuerst) für `DictWriter`Objekte. Beachten Sie, dass komplexe Zahlen geschrieben werden von Eltern umgeben. Dies kann bei anderen Programmen zu Problemen führen CSV-Dateien lesen (vorausgesetzt, sie unterstützen überhaupt komplexe Zahlen).

`csvwriter.writerow`(*Zeile*)

> Schreiben Sie den *Writers* Zeilenparameter entsprechend formatiert in das Dateiobjekt des zum Strom `Dialect`. Gibt den Rückgabewert des Aufrufs an die zurück *write* -Methode des zugrunde liegenden Dateiobjekts.
>
> Geändert in Version 3.5: Unterstützung beliebiger Iterables hinzugefügt.

`csvwriter.writerows`(*Zeilen*)

> Schreiben Sie alle Elemente in *Zeilen* (ein Iterable von Zeilenobjekten *wie* beschrieben oben) in das Dateiobjekt des Writers, formatiert nach dem aktuellen Dialekt.

Writer-Objekte haben das folgende öffentliche Attribut:

`csvwriter.dialect`

> Eine schreibgeschützte Beschreibung des vom Autor verwendeten Dialekts.

DictWriter-Objekte haben die folgende öffentliche Methode:

`DictWriter.writeheader`( )

> Schreiben Sie eine Zeile mit den Feldnamen (wie im Konstruktor angegeben) nach das Dateiobjekt des Autors, formatiert nach dem aktuellen Dialekt. Zurückkehren der Rückgabewert der `csvwriter.writerow()`Aufruf intern verwendet.
>
> Neu in Version 3.2.
>
> Geändert in Version 3.8: `writeheader()`gibt nun auch den von zurückgegebenen Wert zurück das `csvwriter.writerow()`Methode, die es intern verwendet.

Examples

Das einfachste Beispiel zum Lesen einer CSV-Datei:

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

Lesen einer Datei mit einem alternativen Format:

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

```
    for row in reader:
        print(row)
```

Das entsprechende einfachstmögliche Schreibbeispiel lautet:

```
import csv
with open('some.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

Seit `open()`wird verwendet, um eine CSV-Datei zum Lesen zu öffnen, die Datei wird standardmäßig unter Verwendung der Systemvorgabe in Unicode dekodiert Codierung (vgl `locale.getpreferredencoding()`). Um eine Datei zu entschlüsseln Wenn Sie eine andere Codierung verwenden, verwenden Sie die `encoding`Argument von offen:

```
import csv
with open('some.csv', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Dasselbe gilt für das Schreiben in etwas anderem als dem Systemstandard Kodierung: Geben Sie das Kodierungsargument beim Öffnen der Ausgabedatei an.

Registrierung eines neuen Dialekts:

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

Eine etwas fortgeschrittenere Verwendung des Lesegeräts – Abfangen und Melden von Fehlern:

```
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))
```

Und obwohl das Modul das Analysieren von Zeichenfolgen nicht direkt unterstützt, kann dies leicht der Fall sein erledigt:

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

Fußnoten

1 (1 , 2)

> Wenn `newline=''`ist nicht angegeben, Zeilenumbrüche eingebettet in Felder in Anführungszeichen wird nicht richtig interpretiert, und auf Plattformen, die verwenden `\r\n`Bettwäsche auf schreiben Sie ein Extra `\r`wird hinzugefügt werden. Es sollte immer sicher

sein, es anzugeben `newline=''`, da das csv-Modul selbst arbeitet (universelle) Zeilenumbruchbehandlung.

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 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 dialects and formatting parameters, see the Dialects and Formatting Parameters section. 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 dialects 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`(*f*, *fieldnames=None*, *restkey=None*, *restval=None*, *dialect='excel'*, **args*, ***kwds*)

Create an object that operates like a regular reader but maps the information in each row to a `dict` whose keys are given by the optional *fieldnames* parameter.

The *fieldnames* parameter is a sequence. If *fieldnames* is omitted, the values in the first row of file *f* will be used as the fieldnames. Regardless of how the fieldnames are determined, the dictionary preserves their original ordering.

If a row has more fields than fieldnames, the remaining data is put in a list and stored with the fieldname specified by *restkey* (which defaults to `None`). If a non-blank row has fewer fields than fieldnames, the missing values are filled-in with the value of *restval* (which defaults to `None`).

All other optional or keyword arguments are passed to the underlying `reader` instance.

Changed in version 3.6: Returned rows are now of type `OrderedDict`.

Changed in version 3.8: Returned rows are now of type `dict`.

A short usage example:

```
>>> import csv
>>> with open('names.csv', newline='') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Eric Idle
John Cleese

>>> print(row)
{'first_name': 'John', 'last_name': 'Cleese'}
```

class `csv.DictWriter`(*f*, *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 file *f*. 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'`, the default value, 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` class is not optional.

A short usage example:

```
import csv

with open('names.csv', 'w', newline='') 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 whose attributes contain information for how to handle doublequotes, whitespace, delimiters, etc. Due to the lack of a strict CSV specification, different applications produce subtly different CSV data. `Dialect` instances define how `reader` and `writer` instances behave.

All available `Dialect` names are returned by `list_dialects()`, and they can be registered with specific `reader` and `writer` classes through their initializer (`__init__`) functions like this:

```
import csv

with open('students.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile, dialect='unix')
                                 ^^^^^^^^^^^^^^
```

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. Inspecting each column, one of two key criteria will be considered to estimate if the sample contains a header:

- the second through n-th rows contain numeric values
- the second through n-th rows contain strings where at least one value's length differs from that of the putative header of that column.

Twenty rows after the first row are sampled; if more than half of columns + rows meet the criteria, True is returned.

Note

This method is a rough heuristic and may produce both false positives and negatives.

An example for Sniffer use:

```
with open('example.csv', newline='') 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.

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`.

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 (if the object was returned from `reader()`) or a dict (if it is a `DictReader` instance), 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.

Writer Objects

`Writer` objects (`DictWriter` instances and objects returned by the `writer()` function) have the following public methods. A *row* must be an iterable 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`. Return the return value of the call to the *write* method of the underlying file object.

Changed in version 3.5: Added support of arbitrary iterables.

`csvwriter.writerows`(*rows*)

Write all elements in *rows* (an iterable 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) to the writer's file object, formatted according to the current dialect. Return the return value of the `csvwriter.writerow()` call used internally.

New in version 3.2.

Changed in version 3.8: `writeheader()` now also returns the value returned by the `csvwriter.writerow()` method it uses internally.

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.