Reading and Writing CSV Files in Python

realpython.com/python-csv

Let’s face it: you need to get information into and out of your programs through more than just the keyboard and console. Exchanging information through text files is a common way to share info between programs. One of the most popular formats for exchanging data is the CSV format. But how do you use it?

Let’s get one thing clear: you don’t have to (and you won’t) build your own CSV parser from scratch. There are several perfectly acceptable libraries you can use. The Python `csv` library will work for most cases. If your work requires lots of data or numerical analysis, the `pandas` library has CSV parsing capabilities as well, which should handle the rest.

In this article, you’ll learn how to read, process, and parse CSV from text files using Python. You’ll see how CSV files work, learn the all-important `csv` library built into Python, and see how CSV parsing works using the `pandas` library.

So let’s get started!

What Is a CSV File?

A CSV file (Comma Separated Values file) is a type of plain text file that uses specific structuring to arrange tabular data. Because it’s a plain text file, it can contain only actual text data—in other words, printable ASCII or Unicode characters.

The structure of a CSV file is given away by its name. Normally, CSV files use a comma to separate each specific data value. Here’s what that structure looks like:

```
column 1 name,column 2 name, column 3 name
first row data 1,first row data 2,first row data 3
second row data 1,second row data 2,second row data 3
...
```

Notice how each piece of data is separated by a comma. Normally, the first line identifies each piece of data—in other words, the name of a data column. Every subsequent line after that is actual data and is limited only by file size constraints.

In general, the separator character is called a delimiter, and the comma is not the only one used. Other popular delimiters include the tab (`\t`), colon (`:`) and semi-colon (`;`) characters. Properly parsing a CSV file requires us to know which delimiter is being used.

Where Do CSV Files Come From?

CSV files are normally created by programs that handle large amounts of data. They are a convenient way to export data from spreadsheets and databases as well as import or use it in other programs. For example, you might export the results of a data mining program to a CSV file and then import that into a spreadsheet to analyze the data, generate graphs for a presentation, or prepare a report for publication.

CSV files are very easy to work with programmatically. Any language that supports text file input and string manipulation (like Python) can work with CSV files directly.

Parsing CSV Files With Python’s Built-in CSV Library

The `csv` library provides functionality to both read from and write to CSV files. Designed to work out of the box with Excel-generated CSV files, it is easily adapted to work with a variety of CSV formats. The `csv` library contains objects and other code to read, write, and process data from and to CSV files.

Reading CSV Files With `csv`

Reading from a CSV file is done using the `reader` object. The CSV file is opened as a text file with Python’s built-in `open()` function, which returns a file object. This is then passed to the `reader`, which does the heavy lifting.

Here’s the `employee_birthday.txt` file:

```
name,department,birthday month
John Smith,Accounting,November
Erica Meyers,IT,March
```

Here’s code to read it:

```
import csv

with open('employee_birthday.txt') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print(f'Column names are {", ".join(row)}')
            line_count += 1
        else:
            print(f'\t{row[0]} works in the {row[1]} department, and was born in
{row[2]}.')
            line_count += 1
    print(f'Processed {line_count} lines.')
```

This results in the following output:

```
Column names are name, department, birthday month
    John Smith works in the Accounting department, and was born in November.
    Erica Meyers works in the IT department, and was born in March.
Processed 3 lines.
```

Each row returned by the `reader` is a list of `String` elements containing the data found by removing the delimiters. The first row returned contains the column names, which is handled in a special way.

Reading CSV Files Into a Dictionary With `csv`

Rather than deal with a list of individual `String` elements, you can read CSV data directly into a dictionary (technically, an Ordered Dictionary) as well.

Again, our input file, `employee_birthday.txt` is as follows:

```
name,department,birthday month
John Smith,Accounting,November
Erica Meyers,IT,March
```

Here's the code to read it in as a dictionary this time:

```
import csv

with open('employee_birthday.txt', mode='r') as csv_file:
    csv_reader = csv.DictReader(csv_file)
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print(f'Column names are {", ".join(row)}')
            line_count += 1
        print(f'\t{row["name"]} works in the {row["department"]} department, and
was born in {row["birthday month"]}.')
        line_count += 1
    print(f'Processed {line_count} lines.')
```

This results in the same output as before:

```
Column names are name, department, birthday month
    John Smith works in the Accounting department, and was born in November.
    Erica Meyers works in the IT department, and was born in March.
Processed 3 lines.
```

Where did the dictionary keys come from? The first line of the CSV file is assumed to contain the keys to use to build the dictionary. If you don't have these in your CSV file, you should specify your own keys by setting the `fieldnames` optional parameter to a list containing them.

Optional Python CSV `reader` Parameters

The `reader` object can handle different styles of CSV files by specifying additional parameters, some of which are shown below:

- `delimiter` specifies the character used to separate each field. The default is the comma (`','`).
- `quotechar` specifies the character used to surround fields that contain the delimiter character. The default is a double quote (`' " '`).

- `escapechar` specifies the character used to escape the delimiter character, in case quotes aren't used. The default is no escape character.

These parameters deserve some more explanation. Suppose you're working with the following `employee_addresses.txt` file:

```
name,address,date joined
john smith,1132 Anywhere Lane Hoboken NJ, 07030,Jan 4
erica meyers,1234 Smith Lane Hoboken NJ, 07030,March 2
```

This CSV file contains three fields: `name`, `address`, and `date joined`, which are delimited by commas. The problem is that the data for the `address` field also contains a comma to signify the zip code.

There are three different ways to handle this situation:

- **Use a different delimiter**
  That way, the comma can safely be used in the data itself. You use the `delimiter` optional parameter to specify the new delimiter.

- **Wrap the data in quotes**
  The special nature of your chosen delimiter is ignored in quoted strings. Therefore, you can specify the character used for quoting with the `quotechar` optional parameter. As long as that character also doesn't appear in the data, you're fine.

- **Escape the delimiter characters in the data**
  Escape characters work just as they do in format strings, nullifying the interpretation of the character being escaped (in this case, the delimiter). If an escape character is used, it must be specified using the `escapechar` optional parameter.

Writing CSV Files With `csv`

You can also write to a CSV file using a `writer` object and the `.write_row()` method:

```
import csv

with open('employee_file.csv', mode='w') as employee_file:
    employee_writer = csv.writer(employee_file, delimiter=',', quotechar='"',
quoting=csv.QUOTE_MINIMAL)

    employee_writer.writerow(['John Smith', 'Accounting', 'November'])
    employee_writer.writerow(['Erica Meyers', 'IT', 'March'])
```

The `quotechar` optional parameter tells the `writer` which character to use to quote fields when writing. Whether quoting is used or not, however, is determined by the `quoting` optional parameter:

- If `quoting` is set to `csv.QUOTE_MINIMAL`, then `.writerow()` will quote fields only if they contain the `delimiter` or the `quotechar`. This is the default case.
- If `quoting` is set to `csv.QUOTE_ALL`, then `.writerow()` will quote all fields.
- If `quoting` is set to `csv.QUOTE_NONNUMERIC`, then `.writerow()` will quote all fields containing text data and convert all numeric fields to the `float` data type.
- If `quoting` is set to `csv.QUOTE_NONE`, then `.writerow()` will escape delimiters instead of quoting them. In this case, you also must provide a value for the `escapechar` optional parameter.

Reading the file back in plain text shows that the file is created as follows:

```
John Smith,Accounting,November
Erica Meyers,IT,March
```

Writing CSV File From a Dictionary With `csv`

Since you can read our data into a dictionary, it’s only fair that you should be able to write it out from a dictionary as well:

```
import csv

with open('employee_file2.csv', mode='w') as csv_file:
    fieldnames = ['emp_name', 'dept', 'birth_month']
    writer = csv.DictWriter(csv_file, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'emp_name': 'John Smith', 'dept': 'Accounting',
'birth_month': 'November'})
    writer.writerow({'emp_name': 'Erica Meyers', 'dept': 'IT', 'birth_month':
'March'})
```

Unlike `DictReader`, the `fieldnames` parameter is required when writing a dictionary. This makes sense, when you think about it: without a list of `fieldnames`, the `DictWriter` can’t know which keys to use to retrieve values from your dictionaries. It also uses the keys in `fieldnames` to write out the first row as column names.

The code above generates the following output file:

```
emp_name,dept,birth_month
John Smith,Accounting,November
Erica Meyers,IT,March
```