



— FREE Email Series —

 **Python Tricks** 

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

[Get Python Tricks »](#)

 No spam. Unsubscribe any time.

# Reading and Writing CSV Files in Python

by Jon Fincher  69 Comments  [data-science](#) [intermediate](#) [python](#)

[Tweet](#) [Share](#) [Email](#)

## Table of Contents

- [What Is a CSV File?](#)
  - [Where Do CSV Files Come From?](#)
- [Parsing CSV Files With Python's Built-in CSV Library](#)
  - [Reading CSV Files With csv](#)
  - [Reading CSV Files Into a Dictionary With csv](#)

## All Tutorial Topics

[advanced](#) [api](#) [basics](#) [best-practices](#)  
[community](#) [databases](#) [data-science](#)  
[devops](#) [django](#) [docker](#) [flask](#) [front-end](#)  
[intermediate](#) [machine-learning](#) [projects](#)  
[python](#) [testing](#) [tools](#) [web-dev](#)  
[web-scraping](#)

- Optional Python CSV reader Parameters
- Writing CSV Files With csv
- Writing CSV File From a Dictionary With csv
- Parsing CSV Files With the pandas Library
  - Reading CSV Files With pandas
  - Writing CSV Files With pandas
- Conclusion

## Supercharge Your Selenium Python Testing

[Watch Now](#) This tutorial has a related video course created by the author. Watch it together with the written tutorial to deepen your understanding of **Writing CSV Files**

Let's face it: you need to get information into and out of your programs using just the keyboard and console. Exchanging information through files is a way to share info between programs. One of the most popular ways is using the CSV format. But how do you use it?

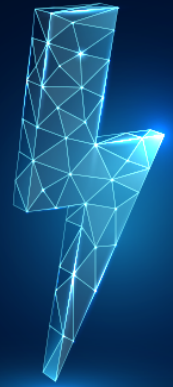
Let's get one thing clear: you don't have to (and you won't) reinvent the wheel from scratch. There are several perfectly acceptable libraries you can use. The Python [csv library](#) will work for most cases. If your work requires lots of data or numerical analysis, the [pandas library](#) has CSV parsing 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 how CSV parsing works using the pandas library.

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```



## Supercharge Your Selenium Python Testing



## Improve Your Python

...with a fresh **Python Trick** code snippet every couple of days:

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

[Send Python Tricks »](#)

[Recommended Video Course](#)


[Reading and Writing CSV Files](#)

Improve Your Python



So let's get started!

**Free Download:** Get a sample chapter from **Python Basics: A Practical Introduction to Python 3** to see how you can go from beginner to intermediate in Python with a complete curriculum, up-to-date for Python 3.8.

 **Take the Quiz:** Test your knowledge with our interactive “Reading and Writing CSV Files in Python” quiz. Upon completion you will receive a score so you can track your learning progress over time:

Take the Quiz »

## 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:

### CSV

```
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:

CSV

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

Here's code to read it:

Python

```
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:

Shell

```
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

## 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:

### CSV

```
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:

### Python

```
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'
              f' was born in {row["birthday month"]}.')
        line_count += 1
    print(f'Processed {line_count} lines.')
```

This results in the same output as before:

### Shell

```
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:

### CSV

```
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:

Python

```
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:

#### CSV

```
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:

#### Python

```
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_mon  
    writer.writerow({'emp_name': 'Erica Meyers', 'dept': 'IT', 'birth_month':
```

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:

#### CSV

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

## Parsing CSV Files With the pandas Library

Of course, the Python CSV library isn't the only game in town. Reading CSV files is possible in [pandas](#) as well. It is highly recommended if you have a lot of data to analyze.

pandas is an open-source Python library that provides high performance data analysis tools and easy to use data structures. pandas is available for all Python installations, but it is a key part of the [Anaconda](#) distribution and works extremely well in [Jupyter notebooks](#) to share data, code, analysis results, visualizations, and narrative text.

Installing pandas and its dependencies in Anaconda is easily done:

#### Shell

```
$ conda install pandas
```

As is using [pip/pipenv](#) for other Python installations:

#### Shell

```
$ pip install pandas
```

We won't delve into the specifics of how pandas works or how to use it. For an in-depth treatment on using pandas to read and analyze large data sets, check out [Shantnu Tiwari's](#) superb article on [working with large Excel files in pandas](#).

## Reading CSV Files With pandas

To show some of the power of pandas CSV capabilities, I've created a slightly more complicated file to read, called `hrdata.csv`. It contains data on company employees:

### CSV

```
Name,Hire Date,Salary,Sick Days remaining
Graham Chapman,03/15/14,50000.00,10
John Cleese,06/01/15,65000.00,8
Eric Idle,05/12/14,45000.00,10
Terry Jones,11/01/13,70000.00,3
Terry Gilliam,08/12/14,48000.00,7
Michael Palin,05/23/13,66000.00,8
```

Reading the CSV into a pandas [DataFrame](#) is quick and straightforward:

### Python

```
import pandas
df = pandas.read_csv('hrdata.csv')
print(df)
```

That's it: three lines of code, and only one of them is doing the actual work.

`pandas.read_csv()` opens, analyzes, and reads the CSV file provided, and stores the data in a [DataFrame](#). Printing the DataFrame results in the following output:

### Shell

```
      Name Hire Date  Salary  Sick Days remaining
0  Graham Chapman  03/15/14  50000.0             10
1    John Cleese   06/01/15  65000.0              8
2    Eric Idle    05/12/14  45000.0             10
```

3	Terry Jones	11/01/13	70000.0	3
4	Terry Gilliam	08/12/14	48000.0	7
5	Michael Palin	05/23/13	66000.0	8

Here are a few points worth noting:

- First, pandas recognized that the first line of the CSV contained column names, and used them automatically. I call this Goodness.
- However, pandas is also using zero-based integer indices in the DataFrame. That's because we didn't tell it what our index should be.
- Further, if you look at the data types of our columns, you'll see pandas has properly converted the Salary and Sick Days remaining columns to numbers, but the Hire Date column is still a String. This is easily confirmed in interactive mode:

Python

>>>

```
>>> print(type(df['Hire Date'][0]))  
<class 'str'>
```

Let's tackle these issues one at a time. To use a different column as the DataFrame index, add the `index_col` optional parameter:

Python

```
import pandas  
df = pandas.read_csv('hrdata.csv', index_col='Name')  
print(df)
```

Now the Name field is our DataFrame index:

### Shell

Name	Hire Date	Salary	Sick Days remaining
Graham Chapman	03/15/14	50000.0	10
John Cleese	06/01/15	65000.0	8
Eric Idle	05/12/14	45000.0	10
Terry Jones	11/01/13	70000.0	3
Terry Gilliam	08/12/14	48000.0	7
Michael Palin	05/23/13	66000.0	8

Next, let's fix the data type of the Hire Date field. You can force pandas to read data as a date with the `parse_dates` optional parameter, which is defined as a list of column names to treat as dates:

### Python

```
import pandas
df = pandas.read_csv('hrdata.csv', index_col='Name', parse_dates=['Hire Date'])
print(df)
```

Notice the difference in the output:

### Shell

Name	Hire Date	Salary	Sick Days remaining
Graham Chapman	2014-03-15	50000.0	10
John Cleese	2015-06-01	65000.0	8
Eric Idle	2014-05-12	45000.0	10
Terry Jones	2013-11-01	70000.0	3
Terry Gilliam	2014-08-12	48000.0	7
Michael Palin	2013-05-23	66000.0	8

The date is now formatted properly, which is easily confirmed in interactive mode:

Python

>>>

```
>>> print(type(df['Hire Date'][0]))
<class 'pandas._libs.tslibs.timestamps.Timestamp'>
```

If your CSV file doesn't have column names in the first line, you can use the `names` optional parameter to provide a list of column names. You can also use this if you want to override the column names provided in the first line. In this case, you must also tell `pandas.read_csv()` to ignore existing column names using the `header=0` optional parameter:

Python

```
import pandas
df = pandas.read_csv('hrdata.csv',
                    index_col='Employee',
                    parse_dates=['Hired'],
                    header=0,
                    names=['Employee', 'Hired', 'Salary', 'Sick Days'])
print(df)
```

Notice that, since the column names changed, the columns specified in the `index_col` and `parse_dates` optional parameters must also be changed. This now results in the following output:

Shell

	Hired	Salary	Sick Days
Employee			
Graham Chapman	2014-03-15	50000.0	10
John Cleese	2015-06-01	65000.0	8
Eric Idle	2014-05-12	45000.0	10

Terry Jones	2013-11-01	70000.0	3
Terry Gilliam	2014-08-12	48000.0	7
Michael Palin	2013-05-23	66000.0	8

## Writing CSV Files With pandas

Of course, if you can't get your data out of pandas again, it doesn't do you much good. Writing a DataFrame to a CSV file is just as easy as reading one in. Let's write the data with the new column names to a new CSV file:

### Python

```
import pandas
df = pandas.read_csv('hrdata.csv',
                    index_col='Employee',
                    parse_dates=['Hired'],
                    header=0,
                    names=['Employee', 'Hired', 'Salary', 'Sick Days'])
df.to_csv('hrdata_modified.csv')
```

The only difference between this code and the reading code above is that the `print(df)` call was replaced with `df.to_csv()`, providing the file name. The new CSV file looks like this:


### Shell

```
Employee,Hired,Salary,Sick Days
Graham Chapman,2014-03-15,50000.0,10
John Cleese,2015-06-01,65000.0,8
Eric Idle,2014-05-12,45000.0,10
Terry Jones,2013-11-01,70000.0,3
Terry Gilliam,2014-08-12,48000.0,7
Michael Palin,2013-05-23,66000.0,8
```

## Conclusion

If you understand the basics of reading CSV files, then you won't ever be caught flat footed when you need to deal with importing data. Most CSV reading, processing, and writing tasks

can be easily handled by the basic `csv` Python library. If you have a lot of data to read and process, the `pandas` library provides quick and easy CSV handling capabilities as well.


 **Take the Quiz:** Test your knowledge with our interactive “Reading and Writing CSV Files in Python” quiz. Upon completion you will receive a score so you can track your learning progress over time:

Take the Quiz »

Are there other ways to parse text files? Of course! Libraries like [ANTLR](#), [PLY](#), and [PlyPlus](#) can all handle heavy-duty parsing, and if simple `String` manipulation won't work, there are always regular expressions.

But those are topics for other articles...

**Free Download:** Get a sample chapter from **Python Basics: A Practical Introduction to Python 3** to see how you can go from beginner to intermediate in Python with a complete curriculum, up-to-date for Python 3.8.

 **Watch Now** This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Reading and Writing CSV Files**





Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Send Me Python Tricks »

## About Jon Fincher

Jon taught Python and Java in two high schools in Washington State. Previously, he was a Program Manager at Microsoft.

» [More about Jon](#)

*Each tutorial at Real Python is created by a team of developers so that it meets our high*

quality standards. The team members who worked on this tutorial are:

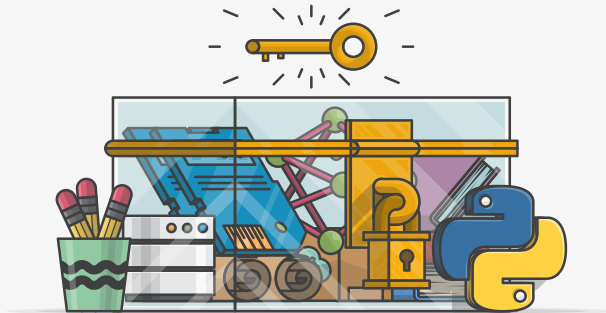
Aldren

Geir Arne

Joanna

Jason

## Master Real-World Python Skills With Unlimited Access to Real Python



**Join us and get access to hundreds of  
tutorials, hands-on video courses, and a  
community of expert Pythonistas:**

[Level Up Your Python Skills »](#)

What Do You Think?

[Tweet](#)[Share](#)[Email](#)

**Real Python Comment Policy:** The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

## Keep Learning

Related Tutorial Categories: [data-science](#) [intermediate](#) [python](#)

Recommended Video Course: [Reading and Writing CSV Files](#)

© 2012–2020 Real Python · [Newsletter](#) · [Podcast](#) · [YouTube](#) · [Twitter](#) · [Facebook](#) · [Instagram](#) ·  
[Python Tutorials](#) · [Search](#) · [Privacy Policy](#) · [Energy Policy](#) · [Advertise](#) · [Contact](#)

♥ Happy Pythoning!