# Structured Text Files: CSV

- `'csv'`: Name of the Python library for using CSV files
- **Lists**: Reading and writing to and from Python lists

```
cin = csv.reader(file)
cout = csv.writer(file) --> cout.writerows(list)
```

- **Dictionaries**: Reading and writing to and from Python dictionaries

```
cin = csv.DictReader(file, fieldnames)
cout = csv.DictWriter(file, headerlist)
cout.writeheader() --> cout.writerows(row-data)
```

© 2014 Cisco and/or its affiliates. All rights reserved. Cisco Confidential

# Structured Text Files: CSV Example

- **Reading CSV**

Reads input from CSV format automatically into list

```
import csv

infile = open('dev_in','r')
csv_in = csv.reader(infile)

for dev in csv_in:

    dev_list.append(dev)
```

- **Writing CSV**

Writes output from list automatically into CSV format

```
import csv

outfile = open('dev_out','w')
csv_out = csv.writer(outfile)

csv_out.writerows(dev_list)
```

© 2014 Cisco and/or its affiliates. All rights reserved. Cisco Confidential

Most common to read lists from CSV

Comma separated value files are read and written using the CSV library. The processes for reading and writing are as follows:

**Writing lists** (using Python list of lists)**:**

- Open the file**:** Open the file for writing as for a regular text file.
- Attach CSV writer: Create a CSV writer and attach it to the file.
- Write rows: Use the CSV writer object to write rows of data from your lists.

**Writing dictionaries** (using Python list of dictionaries)**:**

- Open the file: Open the file for writing as for a regular text file.
- Attach CSV dictionary writer: Create a CSC dictionary writer object and attach it to the file.
- Write header: Use the CSV dictionary writer to write the header – the keys of the dictionary.
- Write rows: Use the CSV dictionary writer to write the rows of dictionary data from your lists.

**Reading lists** (into a Python list of lists)**:**

- Open the file**:** Open the file for reading as for a regular text file.
- Attach CSV reader: Create a CSV reader and attach it to the file.
- You now have a Python list of lists (the actual CSV reader object) which can be used for iteration and for other normal list operations, and the items within the list are themselves normal Python lists which can be used with normal Python list operations.

**Reading dictionaries** (into a Python list of dictionaries)**:**

- Open the file: Open the file for reading as for a regular text file.
- Attach CSV dictionary reader: Create a CSC dictionary reader and attach it to the file.
- You now have a Python list of dictionaries (the actual CSV dictionary reader object), which can be used for iteration and other normal list operations, and the items within the list are dictionaries, which can be used with normal Python dictionary operations.

Here is an example of reading a CSV file. Reading a normal text file is straightforward in Python, since opening a text file for reading provides an object that is iterable:

```
file = open('myfile','r')
for line in file:
    # 'line' must be parsed for specific data items
```

The problem with the above is that 'line' is just an unstructured string, which you will have to parse in some way in order to get data that makes sense.

Reading in the lines from a CSV file is similarly straightforward, with the advantage that the data from each row (line) is automatically parsed and placed into a list.

First you import the csv library for performing functions on CSV files:

```
import csv
```

You will open the file just as you would any other text file:

```
file = open('csv_file', 'r')
```

You then use the csv library to create a CSV object for reading the file, which provides a csv variable which you can use for reading the file:

```
csv_input = csv.reader(file)
```

Iterating through the CSV file looks similar to what is done for a text file, but with the added benefit that the line itself has already been parsed into a list:

```
for items in csv_input:
    # data is already parsed into list
```

The complete example to read in a list of device information:

```
file = open('csv-devices','r')
csv_devices = csv.reader(file)
for device_info in csv_devices:
    # device_info is already parsed into a list of items
```

The main advantage is the ability to immediately have parsed items from each 'row' placed into your Python list, as you iterate through all rows in the file.

Here is an example of writing a Python file. Once again, the file is opened as you would with any normal text file:

```
outfile = open('csv-devices-out', 'w')
csv_devices_out = csv.writer(outfile)
```

For the output example, you will be writing a list of lists. For example, a list of devices, where every individual device is itself a list of device information (device name, IP address, OS-type, username, and password).

The normal operation would then be just to perform the `writerows` function, passing in the list of devices:

```
csv_devices_out.writerows(devices_out_list)
```

In some of the code and lab examples, you will see the use of the `with` statement. The preceding example using `with` would be as follows:

```
with open('csv-devices-out','w') as outfile:
    csv_devices_out = csv.writer(outfile)
    csv_devices_out.writerows(devices_out_list)
```

This example required the same number of lines, but recall that `outfile` will automatically be closed when the code block is completed. And the indentation of the code block improves readability, identifying the lines where the file is being accessed.