Writing a CSV File

Writing a CSV file in python

WE'LL COVER THE FOLLOWING

- Wrapping Up

The csv module also has two methods that you can use to write a CSV file. You can use the **writer** function or the DictWriter class. We'll look at both of these as well. We will be with the writer function. Let's look at a simple example:

```
import csv

def csv_writer(data, path):
    """
    Write data to a CSV file path
    """
    with open(path, "w", newline='') as csv_file:
        writer = csv.writer(csv_file, delimiter=',')
        for line in data:
            writer.writerow(line)

if __name__ == "__main__":
    data = ["first_name,last_name,city".split(","),
            "Tyrese,Hirthe,Strackeport".split(","),
            "Jules,Dicki,Lake Nickolasville".split(","),
            "Dedric,Medhurst,Stiedemannberg".split(",")
            ]
    path = "output.csv"
    csv_writer(data, path)

    # You are welcome to apply the CSV Reading
    # code that you learned in the previous lesson
    # and read the CSV file again.
    # Make sure, it outputs what you just wrote.
```

In the code above, we create a **csv_writer** function that accepts two arguments: data and path. The data is a list of lists that we create at the

bottom of the script. We use a shortened version of the data from the previous

example and split the strings on the comma. This returns a list. So we end up with a nested list that looks like this:

```
[['first_name', 'last_name', 'city'],
 ['Tyrese', 'Hirthe', 'Strackeport'],
 ['Jules', 'Dicki', 'Lake Nickolasville'],
 ['Dedric', 'Medhurst', 'Stiedemannberg']]
```

The **csv_writer** function opens the path that we pass in and creates a csv writer object. Then we loop over the nested list structure and write each line out to disk. Note that we specified what the delimiter should be when we created the writer object. If you want the delimiter to be something besides a comma, this is where you would set it.

Now we're ready to learn how to write a CSV file using the **DictWriter** class! We're going to use the data from the previous version and transform it into a list of dictionaries that we can feed to our hungry DictWriter. Let's take a look:

```
import csv

def csv_dict_writer(path, fieldnames, data):
    """
    Writes a CSV file using DictWriter
    """
    with open(path, "w", newline='') as out_file:
        writer = csv.DictWriter(out_file, delimiter=',', fieldnames=fieldnames)
        writer.writeheader()
        for row in data:
            writer.writerow(row)

if __name__ == "__main__":
    data = ["first_name,last_name,city".split(","),
            "Tyrese,Hirthe,Strackeport".split(","),
            "Jules,Dicki,Lake Nickolasville".split(","),
            "Dedric,Medhurst,Stiedemannberg".split(",")
            ]
    my_list = []
    fieldnames = data[0]
    for values in data[1:]:
        inner_dict = dict(zip(fieldnames, values))
        my_list.append(inner_dict)

    path = "dict_output.csv"
    csv_dict_writer(path, fieldnames, my_list)
```

We will start in the second section first. As you can see, we start out with the nested list structure that we had before. Next we create an empty list and a list that contains the field names, which happens to be the first list inside the nested list. Remember, lists are zero-based, so the first element in a list starts at zero! Next we loop over the nested list construct, starting with the second element:

```
for values in data[1:]:
    inner_dict = dict(zip(fieldnames, values))
    my_list.append(inner_dict)
```

Inside the **for** loop, we use Python builtins to create dictionary. The **zip** method will take two iterators (lists in this case) and turn them into a list of tuples. Here's an example:

```
zip(fieldnames, values)
[('first_name', 'Dedric'), ('last_name', 'Medhurst'), ('city', 'Stiedemannberg')]
```

Now when your wrap that call in **dict**, it turns that list of tuples into a dictionary. Finally we append the dictionary to the list. When the **for** finishes, you'll end up with a data structure that looks like this:

> [{'city': 'Strackeport', 'first_name': 'Tyrese', 'last_name': 'Hirthe'},
>
> : {'city': 'Lake Nickolasville', 'first_name': 'Jules', 'last_name': 'Dicki'},
> {'city': 'Stiedemannberg', 'first_name': 'Dedric', 'last_name': 'Medhurst'}]

At the end of the second session, we call our **csv_dict_writer** function and pass in all the required arguments. Inside the function, we create a DictWriter instance and pass it a file object, a delimiter value and our list of field names. Next we write the field names out to disk and loop over the data one row at a time, writing the data to disk. The DictWriter class also supports the **writerows** method, which we could have used instead of the loop. The **csv.writer** function also supports this functionality.

You may be interested to know that you can also create **Dialects** with the csv

module. This allows you to tell the csv module how to read or write a file in a very explicit manner. If you need this sort of thing because of an oddly formatted file from a client, then you'll find this functionality invaluable.

Wrapping Up

Now you know how to use the csv module to read and write CSV files. There are many websites that put out their data in this format and it is used a lot in the business world. In our next chapter, we will begin learning about the ConfigParser module.