Reading CSV files in Python

We are going to exclusively use the `csv` module built into Python for this task. But first, we will have to import the module as :

```
import csv
```

We have already covered the basics of how to use the `csv` module to read and write into CSV files. If you don't have any idea on using the `csv` module, check out our tutorial on Python CSV: Read and Write CSV files

---

Basic Usage of csv.reader()

Let's look at a basic example of using `csv.reader()` to refresh your existing knowledge.

Example 1: Read CSV files with csv.reader()

Suppose we have a CSV file with the following entries:

```
SN,Name,Contribution
1,Linus Torvalds,Linux Kernel
2,Tim Berners-Lee,World Wide Web
3,Guido van Rossum,Python Programming
```

We can read the contents of the file with the following program:

```
import csv
with open('innovators.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Output

```
['SN', 'Name', 'Contribution']
['1', 'Linus Torvalds', 'Linux Kernel']
['2', 'Tim Berners-Lee', 'World Wide Web']
['3', 'Guido van Rossum', 'Python Programming']
```

Here, we have opened the **innovators.csv** file in reading mode using `open()` function.

To learn more about opening files in Python, visit: Python File Input/Output

Then, the `csv.reader()` is used to read the file, which returns an iterable `reader` object.

The `reader` object is then iterated using a `for` loop to print the contents of each row.

Now, we will look at CSV files with different formats. We will then learn how to customize the `csv.reader()` function to read them.

CSV files with Custom Delimiters

By default, a comma is used as a delimiter in a CSV file. However, some CSV files can use delimiters other than a comma. Few popular ones are `|` and `\t`.

Suppose the **innovators.csv** file in **Example 1** was using **tab** as a delimiter. To read the file, we can pass an additional `delimiter` parameter to the `csv.reader()` function.

Let's take an example.

Example 2: Read CSV file Having Tab Delimiter

```
import csv
with open('innovators.csv', 'r') as file:
    reader = csv.reader(file, delimiter = '\t')
    for row in reader:
        print(row)
```

Output

```
['SN', 'Name', 'Contribution']
```

```
['1', 'Linus Torvalds', 'Linux Kernel']
['2', 'Tim Berners-Lee', 'World Wide Web']
['3', 'Guido van Rossum', 'Python Programming']
```

As we can see, the optional parameter `delimiter = '\t'` helps specify the `reader` object that the CSV file we are reading from, has **tabs** as a delimiter.

CSV files with initial spaces

Some CSV files can have a space character after a delimiter. When we use the default `csv.reader()` function to read these CSV files, we will get spaces in the output as well.

To remove these initial spaces, we need to pass an additional parameter called `skipinitialspace`. Let us look at an example:

Example 3: Read CSV files with initial spaces

Suppose we have a CSV file called **people.csv** with the following content:

```
SN, Name, City
1, John, Washington
2, Eric, Los Angeles
3, Brad, Texas
```

We can read the CSV file as follows:

```
import csv
with open('people.csv', 'r') as csvfile:
    reader = csv.reader(csvfile, skipinitialspace=True)
    for row in reader:
        print(row)
```

Output

```
['SN', 'Name', 'City']
['1', 'John', 'Washington']
['2', 'Eric', 'Los Angeles']
['3', 'Brad', 'Texas']
```

The program is similar to other examples but has an additional `skipinitialspace` parameter which is set to True.

This allows the `reader` object to know that the entries have initial whitespace. As a result, the initial spaces that were present after a delimiter is removed.

CSV files with quotes

Some CSV files can have quotes around each or some of the entries.

Let's take **quotes.csv** as an example, with the following entries:

```
"SN", "Name", "Quotes"
1, Buddha, "What we think we become"
2, Mark Twain, "Never regret anything that made you smile"
3, Oscar Wilde, "Be yourself everyone else is already taken"
```

Using `csv.reader()` in minimal mode will result in output with the quotation marks.

In order to remove them, we will have to use another optional parameter called `quoting`.

Let's look at an example of how to read the above program.

Example 4: Read CSV files with quotes

```
import csv
with open('person1.csv', 'r') as file:
    reader = csv.reader(file, quoting=csv.QUOTE_ALL, skipinitialspace=True)
    for row in reader:
        print(row)
```

Output

```
['SN', 'Name', 'Quotes']
```

```
['1', 'Buddha', 'What we think we become']
['2', 'Mark Twain', 'Never regret anything that made you smile']
['3', 'Oscar Wilde', 'Be yourself everyone else is already taken']
```

As you can see, we have passed `csv.QUOTE_ALL` to the `quoting` parameter. It is a constant defined by the `csv` module.

`csv.QUOTE_ALL` specifies the reader object that all the values in the CSV file are present inside quotation marks.

There are 3 other predefined constants you can pass to the `quoting` parameter:

- `csv.QUOTE_MINIMAL` - Specifies `reader` object that CSV file has quotes around those entries which contain special characters such as **delimiter**, **quotechar** or any of the characters in **lineterminator**.
- `csv.QUOTE_NONNUMERIC` - Specifies the `reader` object that the CSV file has quotes around the non-numeric entries.
- `csv.QUOTE_NONE` - Specifies the reader object that none of the entries have quotes around them.

Dialects in CSV module

Notice in **Example 4** that we have passed multiple parameters (`quoting` and `skipinitialspace`) to the `csv.reader()` function.

This practice is acceptable when dealing with one or two files. But it will make the code more redundant and ugly once we start working with multiple CSV files with similar formats.

As a solution to this, the `csv` module offers `dialect` as an optional parameter.

Dialect helps in grouping together many specific formatting patterns like `delimiter`, `skipinitialspace`, `quoting`, `escapechar` into a single dialect name.

It can then be passed as a parameter to multiple `writer` or `reader` instances.

Example 5: Read CSV files using dialect

Suppose we have a CSV file (**office.csv**) with the following content:

```
"ID"| "Name"| "Email"
"A878"| "Alfonso K. Hamby"| "[emailÂ protected]"
"F854"| "Susanne Briard"| "[emailÂ protected]"
"E833"| "Katja Mauer"| "[emailÂ protected]"
```

The CSV file has initial spaces, quotes around each entry, and uses a `|` delimiter.

Instead of passing three individual formatting patterns, let's look at how to use dialects to read this file.

```
import csv
csv.register_dialect('myDialect',
                     delimiter='|',
                     skipinitialspace=True,
                     quoting=csv.QUOTE_ALL)

with open('office.csv', 'r') as csvfile:
    reader = csv.reader(csvfile, dialect='myDialect')
    for row in reader:
        print(row)
```

Output

```
['ID', 'Name', 'Email']
["A878", 'Alfonso K. Hamby', '[emailÂ protected]']
["F854", 'Susanne Briard', '[emailÂ protected]']
["E833", 'Katja Mauer', '[emailÂ protected]']
```

From this example, we can see that the `csv.register_dialect()` function is used to define a custom dialect. It has the following syntax:

```
csv.register_dialect(name[, dialect[, **fmtparams]])
```

The custom dialect requires a name in the form of a string. Other specifications can be done either by passing a sub-class of `Dialect` class, or by individual formatting patterns as shown in the example.

While creating the reader object, we pass `dialect='myDialect'` to specify that the reader instance must use that particular dialect.

The advantage of using `dialect` is that it makes the program more modular. Notice that we can reuse *'myDialect'* to open other files without having to re-specify the CSV format.

Read CSV files with csv.DictReader()

The objects of a `csv.DictReader()` class can be used to read a CSV file as a dictionary.

Example 6: Python csv.DictReader()

Suppose we have a CSV file (**people.csv**) with the following entries:

Name	Age	Profession
Jack	23	Doctor
Miller	22	Engineer

Let's see how `csv.DictReader()` can be used.

```
import csv
with open("people.csv", 'r') as file:
    csv_file = csv.DictReader(file)
    for row in csv_file:
        print(dict(row))
```

Output

```
{'Name': 'Jack', ' Age': ' 23', ' Profession': ' Doctor'}
{'Name': 'Miller', ' Age': ' 22', ' Profession': ' Engineer'}
```

As we can see, the entries of the first row are the dictionary keys. And, the entries in the other rows are the dictionary values.

Here, *csv_file* is a `csv.DictReader()` object. The object can be iterated over using a `for` loop. The `csv.DictReader()` returned an `OrderedDict` type for each row. That's why we used `dict()` to convert each row to a dictionary.

Notice that we have explicitly used the dict() method to create dictionaries inside the `for` loop.

```
print(dict(row))
```

Note: Starting from Python 3.8, `csv.DictReader()` returns a dictionary for each row, and we do not need to use `dict()` explicitly.

The full syntax of the `csv.DictReader()` class is:

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

To learn more about it in detail, visit: Python csv.DictReader() class

---

Using csv.Sniffer class

The `Sniffer` class is used to deduce the format of a CSV file.

The `Sniffer` class offers two methods:

- `sniff(sample, delimiters=None)` - This function analyses a given sample of the CSV text and returns a `Dialect` subclass that contains all the parameters deduced.

An optional delimiters parameter can be passed as a string containing possible valid delimiter characters.

- `has_header(sample)` - This function returns `True` or `False` based on analyzing whether the sample CSV has the first row as column headers.

Let's look at an example of using these functions:

Example 7: Using csv.Sniffer() to deduce the dialect of CSV files

Suppose we have a CSV file (**office.csv**) with the following content:

```
"ID"| "Name"| "Email"
A878| "Alfonso K. Hamby"| "[emailÂ protected]"
F854| "Susanne Briard"| "[emailÂ protected]"
E833| "Katja Mauer"| "[emailÂ protected]"
```

Let's look at how we can deduce the format of this file using `csv.Sniffer()` class:

```
import csv
with open('office.csv', 'r') as csvfile:
    sample = csvfile.read(64)
    has_header = csv.Sniffer().has_header(sample)
    print(has_header)

    deduced_dialect = csv.Sniffer().sniff(sample)

with open('office.csv', 'r') as csvfile:
    reader = csv.reader(csvfile, deduced_dialect)

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

Output

```
True
['ID', 'Name', 'Email']
['A878', 'Alfonso K. Hamby', '[emailÂ protected]']
['F854', 'Susanne Briard', '[emailÂ protected]']
['E833', 'Katja Mauer', '[emailÂ protected]']
```

As you can see, we read only 64 characters of **office.csv** and stored it in the *sample* variable.

This *sample* was then passed as a parameter to the `Sniffer().has_header()` function. It deduced that the first row must have column headers. Thus, it returned `True` which was then printed out.

Similarly, *sample* was also passed to the `Sniffer().sniff()` function. It returned all the deduced parameters as a `Dialect` subclass which was then stored in the *deduced_dialect* variable.

Later, we re-opened the CSV file and passed the `deduced_dialect` variable as a parameter to `csv.reader()`.

It was correctly able to predict `delimiter`, `quoting` and `skipinitialspace` parameters in the **office.csv** file without us explicitly mentioning them.

Note: The csv module can also be used for other file extensions (like: **.txt**) as long as their contents are in proper structure.

Recommended Reading: Write to CSV Files in Python

Table of Contents