Python CSV: Read and Write CSV Files

(https://www.programiz.com/python-programming/csv)

Python provides a dedicated csv module to work with csv files. The module includes various methods to perform different operations.

However, we first need to import the module using:

```
import csv
```

The csv module provides the csv.reader() function to read a CSV file.

Suppose we have a csv file named people.csv with the following entries.

```
Name,   Age, Profession

Jack,   23,  Doctor

Miller, 22,  Engineer
```

Now, let's read this csv file.

```
import csv

with open('people.csv', 'r') as file:

  reader = csv.reader(file)

  for row in reader:

    print(row)
```

Output

```
['Name', 'Age', 'Profession']

['Jack', '23', 'Doctor']

['Miller', '22', 'Engineer']
```

Here, we have opened the **people.csv** file in reading mode.

Write to CSV Files with Python

The csv module provides the csv.writer() function to write to a CSV file.

Let's look at an example.

```
import csv

with open('protagonist.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(["SN", "Movie", "Protagonist"])
    writer.writerow([1, "Lord of the Rings", "Frodo Baggins"])
    writer.writerow([2, "Harry Potter", "Harry Potter"])
```

When we run the above program, a **protagonist.csv** file is created with the following content:

```
SN,Movie,Protagonist
1,Lord of the Rings,Frodo Baggins
2,Harry Potter,Harry Potter
```

In this example, we have created the CSV file named protagonist.csv in the writing mode.We then used the csv.writer() function to write to the file.

Here,

- `writer.writerow(["SN", "Movie", "Protagonist"])` writes the header row with column names to the CSV file.
- `writer.writerow([1, "Lord of the Rings", "Frodo Baggins"])` writes the first data row to the CSV file.
- `writer.writerow([2, "Harry Potter", "Harry Potter"])` writes the second data row to the CSV file.

Using Python Pandas to Handle CSV Files

Pandas is a popular data science library in Python for data manipulation and analysis.

If we are working with huge chunks of data, it's better to use pandas to handle CSV files for ease and efficiency.

To read the CSV file using pandas, we can use the read_csv() function.

```
import pandas as pd

pd.read_csv("people.csv")
```

Here, the program reads people.csv from the current directory.

To write to a CSV file, we need to use the to_csv() function of a DataFrame.

```
import pandas as pd


# creating a data frame
df = pd.DataFrame([['Jack', 24], ['Rose', 22]], columns = ['Name', 'Age'])


# writing data frame to a CSV file
df.to_csv('person.csv')
```

Here, we have created a DataFrame using the pd.DataFrame() method. Then, the to_csv() function for this object is called, to write into person.csv.

Part2: Reading CSV files in Python

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.

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.

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.

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"| "alfonsokhamby@rhyta.com"
"F854"| "Susanne Briard"| "susannebriard@armyspy.com"
"E833"| "Katja Mauer"| "kmauer@jadoop.com"
```

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', 'alfonsokhamby@rhyta.com']
["F854", 'Susanne Briard', 'susannebriard@armyspy.com']
["E833", 'Katja Mauer', 'kmauer@jadoop.com']
```

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)
```

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"| "alfonsokhamby@rhyta.com"
F854| "Susanne Briard"| "susannebriard@armyspy.com"
E833| "Katja Mauer"| "kmauer@jadoop.com"
```

- 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', 'alfonsokhamby@rhyta.com']
['F854', 'Susanne Briard', 'susannebriard@armyspy.com']
['E833', 'Katja Mauer', 'kmauer@jadoop.com']
```

- 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.

Writing CSV files in Python

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

Example 1: Write into CSV files with csv.writer()

Suppose we want to write 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
```

Here's how we do it.

```
import csv
with open('innovators.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(["SN", "Name", "Contribution"])
    writer.writerow([1, "Linus Torvalds", "Linux Kernel"])
    writer.writerow([2, "Tim Berners-Lee", "World Wide Web"])
    writer.writerow([3, "Guido van Rossum", "Python Programming"])
```

When we run the above program, an **innovators.csv** file is created in the current working directory with the given entries.

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

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

Next, the csv.writer() function is used to create a writer object.
The writer.writerow() function is then used to write single rows to the CSV file.

Example 2: Writing Multiple Rows with writerows()

If we need to write the contents of the 2-dimensional list to a CSV file, here's how we can do it.

```
import csv
row_list = [["SN", "Name", "Contribution"],
            [1, "Linus Torvalds", "Linux Kernel"],
```

```
        [2, "Tim Berners-Lee", "World Wide Web"],
        [3, "Guido van Rossum", "Python Programming"]]
with open('protagonist.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerows(row_list)
```

The output of the program is the same as in **Example 1**.

Here, our 2-dimensional list is passed to the writer.writerows() function to write the content of the list to the CSV file.

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 we want to use | as a delimiter in the **innovators.csv** file of **Example 1**. To write this file, we can pass an additional delimiter parameter to the csv.writer() function.

Example 3: Write CSV File Having Pipe Delimiter

```
import csv
data_list = [["SN", "Name", "Contribution"],
        [1, "Linus Torvalds", "Linux Kernel"],
        [2, "Tim Berners-Lee", "World Wide Web"],
        [3, "Guido van Rossum", "Python Programming"]]
with open('innovators.csv', 'w', newline='') as file:
    writer = csv.writer(file, delimiter='|')
    writer.writerows(data_list)
```

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 = '|' helps specify the writer object that the CSV file should have | as a delimiter.

CSV files with Quotes

Some CSV files 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.writer() by default will not add these quotes to the entries.

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

Let's take an example of how quoting can be used around the non-numeric values and ; as delimiters.

Example 4: Write CSV files with quotes

```
import csv
row_list = [
  ["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"]
]
with open('quotes.csv', 'w', newline='') as file:
  writer = csv.writer(file, quoting=csv.QUOTE_NONNUMERIC, delimiter=';')
  writer.writerows(row_list)
```

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"
```

Here, the **quotes.csv** file is created in the working directory with the above entries.

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

`csv.QUOTE_NONNUMERIC` specifies the writer object that quotes should be added around the non-numeric entries.

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

- `csv.QUOTE_ALL` - Specifies the writer object to write CSV file with quotes around all the entries.
- `csv.QUOTE_MINIMAL` - Specifies the writer object to only quote those fields which contain special characters (**delimiter**, **quotechar** or any characters in **lineterminator**)
- `csv.QUOTE_NONE` - Specifies the writer object that none of the entries should be quoted. It is the default value.

CSV files with custom quoting character

We can also write CSV files with custom quoting characters. For that, we will have to use an optional parameter called quotechar.

Let's take an example of writing **quotes.csv** file in **Example 4**, but with * as the quoting character.

Example 5: Writing CSV files with custom quoting character

```
import csv

row_list = [

  ["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"]

]

with open('quotes.csv', 'w', newline='') as file:
```

```
writer = csv.writer(file, quoting=csv.QUOTE_NONNUMERIC,
                    delimiter=';', quotechar='*')
writer.writerows(row_list)
```

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*
```

Here, we can see that quotechar='*' parameter instructs the writer object to use * as quote for all non-numeric values.

Dialects in CSV module

Notice in **Example 5** that we have passed multiple parameters (quoting, delimiter and quotechar) to the csv.writer() 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 6: Write CSV file using dialect

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

```
"ID"|"Name"|"Email"
"A878"|"Alfonso K. Hamby"|"alfonsokhamby@rhyta.com"
"F854"|"Susanne Briard"|"susannebriard@armyspy.com"
"E833"|"Katja Mauer"|"kmauer@jadoop.com"
```

The CSV file has quotes around each entry and uses | as a delimiter.

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

```
import csv
row_list = [
  ["ID", "Name", "Email"],
  ["A878", "Alfonso K. Hamby", "alfonsokhamby@rhyta.com"],
  ["F854", "Susanne Briard", "susannebriard@armyspy.com"],
  ["E833", "Katja Mauer", "kmauer@jadoop.com"]
]
csv.register_dialect('myDialect',
            delimiter='|',
            quoting=csv.QUOTE_ALL)
with open('office.csv', 'w', newline='') as file:
  writer = csv.writer(file, dialect='myDialect')
  writer.writerows(row_list)
```

Output

```
"ID"|"Name"|"Email"
"A878"|"Alfonso K. Hamby"|"alfonsokhamby@rhyta.com"
"F854"|"Susanne Briard"|"susannebriard@armyspy.com"
"E833"|"Katja Mauer"|"kmauer@jadoop.com"
```

Here, **office.csv** is created in the working directory with the above contents.

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

```
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 the Dialect class, or by individual formatting patterns as shown in the example.

While creating the writer object, we pass dialect='myDialect' to specify that the writer 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 write other CSV files without having to re-specify the CSV format.

Write CSV files with csv.DictWriter()

The objects of csv.DictWriter() class can be used to write to a CSV file from a Python dictionary.

The minimal syntax of the csv.DictWriter() class is:

```
csv.DictWriter(file, fieldnames)
```

Here,

- file - CSV file where we want to write to
- fieldnames - a list object which should contain the column headers specifying the order in which data should be written in the CSV file

Example 7: Python csv.DictWriter()

```
import csv


with open('players.csv', 'w', newline='') as file:
    fieldnames = ['player_name', 'fide_rating']
    writer = csv.DictWriter(file, fieldnames=fieldnames)


    writer.writeheader()
    writer.writerow({'player_name': 'Magnus Carlsen', 'fide_rating': 2870})
    writer.writerow({'player_name': 'Fabiano Caruana', 'fide_rating': 2822})
    writer.writerow({'player_name': 'Ding Liren', 'fide_rating': 2801})
```

Output

The program creates a **players.csv** file with the following entries:

```
player_name,fide_rating
Magnus Carlsen,2870
```

```
Fabiano Caruana,2822

Ding Liren,2801
```

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

```
csv.DictWriter(f, fieldnames, restval='', extrasaction='raise', dialect='excel', *args, **kwds)
```

CSV files with lineterminator

A lineterminator is a string used to terminate lines produced by writer objects. The default value is \r\n. You can change its value by passing any string as a lineterminator parameter.

However, the reader object only recognizes \n or \r as lineterminator values. So using other characters as line terminators is highly discouraged.

---

doublequote & escapechar in CSV module

In order to separate delimiter characters in the entries, the csv module by default quotes the entries using quotation marks.

So, if you had an entry: He is a strong, healthy man, it will be written as: "He is a strong, healthy man".

Similarly, the csv module uses double quotes in order to escape the quote character present in the entries by default.

If you had an entry: Go to "programiz.com", it would be written as: "Go to ""programiz.com""".

Here, we can see that each " is followed by a " to escape the previous one.

---

doublequote

It handles how quotechar present in the entry themselves are quoted. When True, the quoting character is doubled and when False, the escapechar is used as a prefix to the quotechar. By default its value is True.

---

escapechar

escapechar parameter is a string to escape the delimiter if quoting is set to csv.QUOTE_NONE and quotechar if doublequote is False. Its default value is None.

Example 8: Using escapechar in csv writer

```
import csv
row_list = [
  ['Book', 'Quote'],
  ['Lord of the Rings',
    '"All we have to decide is what to do with the time that is given us."'],
  ['Harry Potter', '"It matters not what someone is born, but what they grow to be."']
]
with open('book.csv', 'w', newline='') as file:
  writer = csv.writer(file, escapechar='/', quoting=csv.QUOTE_NONE)
  writer.writerows(row_list)
```

Output

```
Book,Quote
Lord of the Rings,/"All we have to decide is what to do with the time that is given us./"
Harry Potter,/"It matters not what someone is born/, but what they grow to be./"
```

Here, we can see that / is prefix to all the " and , because we specified `quoting=csv.QUOTE_NONE`.

If it wasn't defined, then, the output would be:

```
Book,Quote
Lord of the Rings,"""All we have to decide is what to do with the time that is given us."""
Harry Potter,"""It matters not what someone is born, but what they grow to be."""
```

Since we allow quoting, the entries with special characters(" in this case) are double-quoted. The entries with delimiter are also enclosed within quote characters.(Starting and closing quote characters)

The remaining quote characters are to escape the actual " present as part of the string, so that they are not interpreted as quotechar.

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

Writing Dictionaries to CSV Files in Python

In Python, we use csv.DictWriter() from the csv module to write data stored in dictionaries to a CSV file. Each dictionary key maps to a column header, making it easy for us to write structured data.

Syntax:

```
csv.DictWriter(csvfile, fieldnames, restval='', extrasaction='raise', dialect='excel', *args, **kwds)
```

Parameters:

- **csvfile:** File object opened in write ('w') mode.
- **fieldnames:** List of dictionary keys to be used as column headers.
- **restval:** Default value for missing keys.
- **extrasaction:** What to do if a key not in fieldnames is found ('raise' or 'ignore').
- **dialect:** Formatting style (default is 'excel').
- ***args, **kwds:** Additional formatting options.

Methods:

- **writeheader():** Writes the fieldnames (header row) to the CSV file.
- **writerows(mydict):** Writes multiple dictionaries (rows) to the CSV file.

Example:

```
import csv


# Open the file in write mode
with open('students.csv', mode='w', newline='') as file:
    fieldnames = ['Name', 'Age', 'City']
    writer = csv.DictWriter(file, fieldnames=fieldnames)
```

```
  writer.writeheader()  # Write header row
  writer.writerows([
    {'Name': 'Amit', 'Age': 20, 'City': 'New York'},
    {'Name': 'Mohit', 'Age': 23, 'City': 'Los Angeles'},
    {'Name': 'Ritik', 'Age': 26, 'City': 'Chicago'}
  ])
```

Output:

After running the code, the students.csv file will contain the following data:

```
Name,Age,City
Amit,20,New York
Mohit,23,Los Angeles
Ritik,26,Chicago
```

In this example, we use DictWriter to create a CSV file, write a header, and add student records using dictionary key-value pairs.

Adding new columns to CSV data in Python

Adding a new column involves creating additional data based on the existing columns or with new data. For instance, we might want to add a column that calculates each employee's age next year.

Here's how you can add a new column:

```
# Add a new column 'Age Next Year'
for row in data:
  row['age_next_year'] = int(row['Age']) + 1

# Display the data with the new column
for row in data:
  print(row)
```

This loop iterates over each row in the data list. Each row adds a new key-value pair where the key is ‘age_next_year’ and the value is the current age incremented by one. This creates a new column in the dataset.

How to analyze and modify CSV files in Python

Data manipulation and analysis are essential skills for any data analyst or scientist. In this section, we will look at basic data manipulation in Python.

The following are some data analysis things we can do with the CSV module in Python

Filtering rows in CSV data using Python

Filtering data involves selecting rows that meet certain criteria. For example, we might want to filter rows where the employees’ ages are greater than 25.

Here’s how we can filter the data for that:

```
filtered_data = [row for row in data if int(row['Age']) > 25]


# Display the filtered data
for row in filtered_data:
    print(row)
```