Chapter 20
Working with CSV Files

20.1 Introduction

This chapter introduces a module that supports the generation of CSV (or Comma Separated Values) files.

20.2 CSV Files

The CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases. However, CSV is not a precise standard with multiple different applications having different conventions and specific standards.

The Python `csv` module implements classes to read and write tabular data in CSV format. As part of this it supports the concept of a dialect which is a CSV format used by a specific application or suite of programs, for example, it supports an Excel dialect.

This allows programmers to say, "write this data in the format preferred by Excel," or "read data from this file which was generated by Excel," without knowing the precise details of the CSV format used by Excel.

Programmers can also describe the CSV dialects understood by other applications or define their own special-purpose CSV dialects.

The `csv` module provides a range of functions including:

- `csv.reader (csvfile, dialect='excel', **fmtparams)` Returns a reader object which will iterate over lines in the given csvfile. An optional dialect parameter can be given. This may be an instance of a subclass of the `Dialect` class or one of the strings returned by the `list_dialects()` function. The other optional `fmtparams` keyword arguments can be given to override individual formatting parameters in the current dialect.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_20

- `csv.writer (csvfile, dialect='excel', **fmtparams)` Returns a writer object responsible for converting the user's data into delimited strings on the given csvfile. An optional dialect parameter provided. The `fmtparams` keyword arguments can be given to override individual formatting parameters in the current dialect.
- `csv.list_dialects()` Return the names of all registered dialects. For example on a Mac OS X the default list of dialects is ['excel', 'excel-tab', 'unix'].

20.2.1 The CSV Writer Class

A CSV Writer is obtained from the `csv.writer()` function. The `csvwriter` supports two methods used to write data to the CSV file:

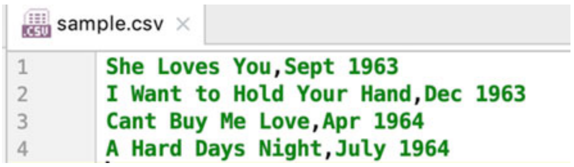
- `csvwriter.writerow(row)` Write the row parameter to the writer's file object, formatted according to the current dialect.
- `csvwriter.writerows(rows)` Write all elements in rows (an iterable of row objects as described above) to the writer's file object, formatted according to the current dialect.
- Writer objects also have the following public attribute:
- `csvwriter.dialect` A read-only description of the dialect in use by the writer.

The following program illustrates a simple use of the `csv` module which creates a file called `sample.csv`.

As we have not specified a dialect, the default 'excel' dialect will be used. The `writerow()` method is used to write each comma separate list of strings to the CSV file.

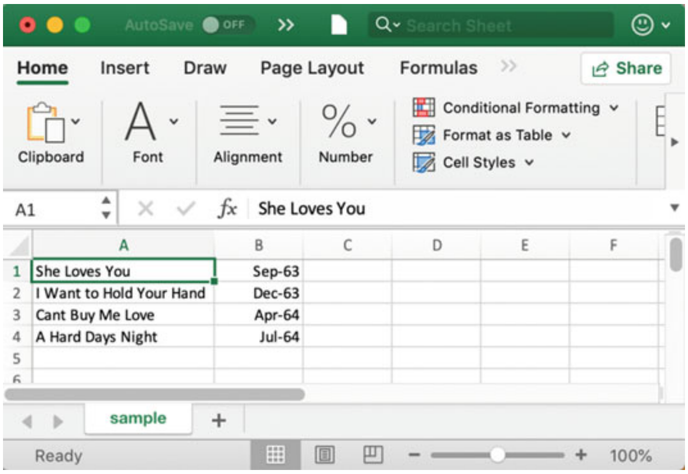
```
print('Crearting CSV file')
with open('sample.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(['She Loves You', 'Sept 1963'])
    writer.writerow(['I Want to Hold Your Hand', 'Dec 1963'])
    writer.writerow(['Cant Buy Me Love', 'Apr 1964'])
    writer.writerow(['A Hard Days Night', 'July 1964'])
```

The resulting file can be viewed as shown below:



```
She Loves You,Sept 1963
I Want to Hold Your Hand,Dec 1963
Cant Buy Me Love,Apr 1964
A Hard Days Night,July 1964
```

However, as it is a CSV file, we can also open it in Excel:



	A	B	C	D	E	F
1	She Loves You	Sep-63				
2	I Want to Hold Your Hand	Dec-63				
3	Cant Buy Me Love	Apr-64				
4	A Hard Days Night	Jul-64				
5						
6						

20.2.2 *The CSV Reader Class*

A CSV Reader object is obtained from the `csv.reader()` function. It implements the iteration protocol.

If a csv reader object is used with a `for` loop then each time round the loop it supplies the next *row* from the CSV file as a list, parsed according to the current CSV dialect.

Reader objects also have the following public attributes:

- `csvreader.dialect` A read-only description of the dialect in use by the parser.
- `csvreader.line_num` The number of lines read from the source iterator. This is not the same as the number of records returned, as records can span multiple lines.

The following provides a very simple example of reading a CSV file using a csv reader object:

```
print('Starting to read csv file')
with open('sample.csv', newline='') as csvfile:
    reader = csv.reader(csvfile)
    for row in reader:
        print(*row, sep=', ')

print('Done Reading')
```

The output from this program, based on the sample.csv file created earlier is:

```
Starting to read csv file
She Loves You, Sept 1963
I Want to Hold Your Hand, Dec 1963
Cant Buy Me Love, Apr 1964
A Hard Days Night, July 1964
Done Reading
```

20.2.3 The CSV DictWriter Class

In many cases the first row of a CSV file contains a set of names (or keys) that define the fields within the rest of the CSV. That is the first row gives meaning to the columns and the data held in the rest of the CSV file. It is therefore very useful to capture this information and to structure the data written to a CSV file or loaded from a CSV file based on the keys in the first row.

The `csv.DictWriter` returns an object that can be used to write values into the CSV file based on the use of such named columns. The file to be used with the `DictWriter` is provided when the class is instantiated.

```
import csv

with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name', 'result']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerow({'first_name': 'John',
                     'last_name': 'Smith',
                     'result' : 54})
    writer.writerow({'first_name': 'Jane',
                     'last_name': 'Lewis',
                     'result': 63})
    writer.writerow({'first_name': 'Chris',
                     'last_name': 'Davies',
                     'result' : 72})
```

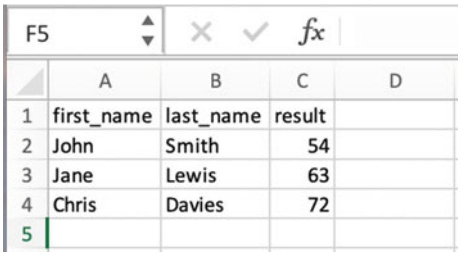
Note that when the `DictWriter` is created a list of the keys must be provided that are used for the columns in the CSV file.

The method `writeheader()` is then used to write the header row out to the CSV file.

The method `writerow()` takes a dictionary object that has keys based on the keys defined for the `DictWriter`. These are then used to write data out to the CSV (note the order of the keys in the dictionary is not important).

In the above example code the result of this is that a new file called `names.csv` is created which can be opened in Excel:

Of course, as this is a CSV file it can also be opened in a plain text editor as well.



	A	B	C	D
1	first_name	last_name	result	
2	John	Smith	54	
3	Jane	Lewis	63	
4	Chris	Davies	72	
5				

20.2.4 The CSV DictReader Class

As well as the `csv.DictWriter` there is a `csv.DictReader`. The file to be used with the `DictReader` is provided when the class is instantiated. As with the `DictReader` the `DictWriter` class takes a list of keys used to define the columns in the CSV file. If the headings to be used for the first row can be provided although this is optional (if a set of keys are not provided, then the values in the first row of the CSV file will be used as the `fieldnames`).

The `DictReader` class provides several useful features including the `fieldnames` property that contains a list of the keys/headings for the CSV file as defined by the first row of the file.

The `DictReader` class also implements the *iteration* protocol and thus it can be used in a for loop in which each row (after the first row) is returned in turn as a dictionary. The dictionary object representing each row can then be used to access each column value based on the keys defined in the first row.

An example is shown below for the CSV file created earlier:

```
import csv

print('Starting to read dict CSV example')

with open('names.csv', newline='') as csvfile:
    reader = csv.DictReader(csvfile)
    for heading in reader.fieldnames:
        print(heading, end=' ')

    print('\n------------------------------')

    for row in reader:
        print(row['first_name'], row['last_name'],
row['result'])

print('Done')
```

This generates the following output:

```
Starting to read dict CSV example
first_name last_name result
------------------------------
John Smith 54
Jane Lewis 63
Chris Davies 72
Done
```

20.3 Online Resources

See the following online resources for information on the topics in this chapter:

- https://docs.python.org/3/library/csv.html for the Python Standard documentation on CSV file reading and writing.
- https://pymotw.com/3/csv/index.html for the Python Module of the Week page on CSV files.
- https://pythonprogramming.net/reading-csv-files-python-3 for a tutorial on reading CSV files.

20.4 Exercises

In this exercise you will create a CSV file based on a set of transactions stored in a current account.

1. To do this first define a new `Account` class to represent a type of bank account.
2. When the class is instantiated you should provide the account number, the name of the account holder, an opening balance and the type of account (which can be a string representing 'current', 'deposit' or 'investment' etc.). This means that there must be an `__init__` method and you will need to store the data within the object.
3. Provide three instance methods for the Account; `deposit(amount)`, `withdraw(amount)` and `get_balance()`. The behaviour of these methods should be as expected, deposit will increase the balance, `withdraw` will decrease the balance and `get_balance()` returns the current balance.

Your `Account` class should also keep a history of the transactions it is involved in.

A *Transaction* is a record of a deposit or withdrawal along with an amount. Note that the initial amount in an account can be treated as an initial deposit.

The history could be implemented as a *list* containing an ordered sequence to transactions. A Transaction itself could be defined by a class with an action (deposit or withdrawal) and an amount.

Each time a withdrawal or a deposit is made a new transaction record should be added to a transaction history list.

Next provide a function (which could be called something like `write_account_transactions_to_csv()`) that can take an account and then write each of the transactions it holds out to a CSV file, with each transaction type and the transaction amount separated by a comma.

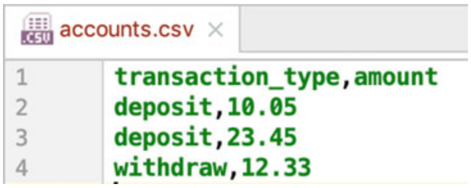
The following sample application illustrates how this function might be used:

```
print('Starting')
acc = accounts.CurrentAccount('123', 'John', 10.05, 100.0)
acc.deposit(23.45)
acc.withdraw(12.33)

print('Writing Account Transactions')
write_account_transaction_to_csv('accounts.csv', acc)

print('Done')
```

The contents of the CSV file would then be:


accounts.csv

```
transaction_type,amount
deposit,10.05
deposit,23.45
withdraw,12.33
```