Python 3: Structured text (CSV) files

Bruce Beckles mbb10@cam.ac.uk

Bob Dowling rjd4@cam.ac.uk

29 October 2012

Prerequisites

This self-paced course assumes that you have a knowledge of Python 3 equivalent to having completed one or other of

- Python 3: Introduction for Absolute Beginners, or
- Python 3: Introduction for Those with Programming Experience

Some experience beyond these courses is always useful but no other course is assumed.

The course also assumes that you know how to use a Unix text editor (gedit, emacs, vi, …).

Facilities for this session

The computers in this room have been prepared for these self-paced courses. They are already logged in with course IDs and have home directories specially prepared. Please do not log in under any other ID.

At the end of this session the home directories will be cleared. Any files you leave in them will be deleted. Please copy any files you want to keep.

The home directories contain a number of subdirectories one for each topic.

For this topic please enter directory `csv`. All work will be completed there:

```
$ cd csv
$ pwd
/home/x250/csv
$
```

These courses are held in a room with two demonstrators. If you get stuck or confused, or if you just have a question raised by something you read, please ask!

These handouts and the prepared folders to go with them can be downloaded from `www.ucs.cam.ac.uk/docs/course-notes/unix-courses/pythontopics`

You are welcome to annotate and keep this handout.

The formal Python 3 documentation for the topics covered here can be found online at `docs.python.org/release/3.2.3/library/csv.html`

Table of Contents

Notation

Warnings

!

Warnings are marked like this. These sections are used to highlight common mistakes or misconceptions.

Exercises

Exercise 0

Exercises are marked like this. You are expected to complete all exercises. Some of them do depend on previous exercises being successfully completed.
An indication is given as to how long we expect the exercise to take. Do not panic if you take longer than this. If you are stuck, ask a demonstrator.
Exercises marked with an asterisk (*) are optional and you should only do these if you have time.

Input and output

Material appearing in a terminal is presented like this:

```
$ more lorem.txt
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
--More--(44%)
```

The material you type is presented like this: **`ls`**. (Bold face, typewriter font.)

The material the computer responds with is presented like this: "`Lorem ipsum`". (Typewriter font again but in a normal face.)

Keys on the keyboard

Keys on the keyboard will be shown as the symbol on the keyboard surrounded by square brackets, so the "A key" will be written "[A]". Note that the return key (pressed at the end of every line of commands) is written "[↵]", the shift key as "[⇧]", and the tab key as "[↹]". Pressing more than one key at the same time (such as pressing the shift key down while pressing the A key) will be written as "[⇧]+[A]". Note that pressing [A] generates the lower case letter "a". To get the upper case letter "A" you need to press [⇧]+[A].

Content of files

The content[1] of files (with a comment) will be shown like this:

```
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur.                                This is a comment about the line.
```

1 The example text here is the famous "lorem ipsum" dummy text used in the printing and typesetting industry. It dates back to the 1500s. See http://www.lipsum.com/ for more information.

What’s in this course

In the introductory courses we warn you not to use the `split()` method of strings to split lines on, for example, commas. We tell you that Python has a specialized module to do this better than you ever could. This topic is about that module.

This topic introduces the `csv` module but does not cover every last detail.

The online Python documentation for this module is at:
`http://docs.python.org/py3k/library/csv.html`

After importing the module you can get help:

```
>>> help(csv)
```

What is a CSV file?

We've already met the limitations of the `split()` method of strings for handling input.

Exercise 1

(a) Look at the files `data1.csv`, `data2.csv`, `data3.csv`, `data4.csv` to see different ways that a spreadsheet can export the same data as text.
(b) Look at the files `weird1.csv`, `weird2.csv`, `weird3.csv`, `weird4.csv`, `weird5.csv`, `weird6.csv` to see some examples of awkward cases.

These files are all examples of "**Comma Separated Value (CSV) files**".

These files are typically created by exporting from spreadsheets or databases, often for import into other spreadsheets or databases.

CSV files are files where each line has the same structure, consisting of a number of values (called "**fields**") separated by some specified character (or sequence of characters), typically a comma (hence the name "comma separated values"). The character (or characters) that separates the fields is called a "**delimiter**".

There may be spaces after (or before) the delimiter to separate the fields, or the fields may be "padded out" with spaces to make them all have the same number of characters. (The number of characters in a field is called the field's "**width**".)

Obviously, we could write our own functions for making sense of (parsing) every arrangement of data we come across, but there is an easier way for this large class of text files. We use one of the standard Python modules: the **`csv`** module.

Compare `data1.csv` and `data2.csv`. The file `data2.csv` doesn't even use commas but TABs as its delimiter. Also, in CSV files, data with spaces in it is often "**quoted**" (surrounded by quotation marks) to make clear that it is one single item of data and should be treated as such. In Python this is not necessary unless you are using a space as your delimiter, but you will often find that programs that produce CSV files automatically quote data with spaces in it.

Look at `data4.csv`. If you want, you can quote all the data in the file, or all the text data. Python doesn't mind if you quote data even when it is not strictly necessary.

Look at von Hayek's entry in `weird4.csv`. If your data contains special characters, such as the delimiter or a new line ('`\n`') character, then you will need to quote that data or Python will get confused when it reads the CSV file.

The Python `csv` module

The Python `csv` module provides ways to read and write CSV files. We load the module in the usual way using `import`:

```
>>> import csv
```

And, once the module has been loaded, we can get help on it using `help()`:

```
>>> help(csv)
```

As well as functions and types of objects ("**classes**") for working with CSV files, the module also provides some constants that are used to set certain options for its functions. We will meet some of these options later.

If you have already done the topic on exceptions ("Python 3: Handling errors"), then it is worth noting that the `csv` module also defines an exception that is used when something goes wrong with any of its functions. This exception is "**`Error`**", but as it is defined in the `csv` module, you would normally refer to it by prefixing it with "**`csv.`**", e.g.

```
try:
    Do something using functions from the csv module
except csv.Error:
    print('Problem with CSV file!')
```

Opening files for use with the `csv` module

As we saw in the introductory courses, Python usually reads a text file a line at a time. Python knows when a line ends because each line ends with an "**end of line**" (**EOL**) indicator[2]. On Unix and Linux the EOL indicator is the new line character ('`\n`'); on Windows it is actually two characters, one of which is the new line character. And, as we saw earlier, the fields in a CSV file might contain new line characters (e.g. in the `weird4.csv` file in Exercise 1).

So that the `csv` module can handle CSV files correctly regardless of the operating system on which the files were created, and regardless of whether or not the fields of the file contain new line characters, we need to tell Python ***not*** to do any special handling of the EOL indicator for CSV files. We do this by using a special option when we open the file: `newline=''`.

For example, we would open a CSV file for reading like this:

```
>>> input_file = open('data1.csv','r',newline='')
```

…and we would open a new CSV file for writing like this:

```
>>> output_file = open('output.csv','w',newline='')
```

! In the arguments we give the `open()` function, the "`newline=''`" option always comes *after* the name of the file and the mode we in which we want to open the file.

2 The documentation for the `csv` module refers to the EOL indicator as the "**line terminator**" or the "string used to terminate lines".

Reading from a CSV file

So how do we read from a CSV file?

First we open the file (in text mode) for reading (making sure we give `open()` the `newline=''` option). Then we create a special type of object to access the CSV file. This is the `reader` object, which is defined in the `csv` module, and which we create using the `reader()` function. The `reader` object is an iterable that gives us access to each line of the CSV file as a list of fields. As the `reader` object is an iterable, we can use `next()` directly on it to read the next line of the CSV file, or we can treat it like a list in a `for` loop to read all the lines of the file (as lists of the file's fields). When we've finished reading from the file we delete the `reader` object and then close the file.

There are various options we can give the `reader()` function when we create the `reader` object to deal with CSV files that use delimiters other than commas, that use padding or which quote some (or all) of their fields. We'll meet these options later. Note that normally the `reader` object returns the fields of the CSV file to us as *strings* (regardless of what they actually contain), although there is an option we'll meet later that changes this behaviour.

So let's try reading from a CSV file. Start an interactive Python 3 session (if you haven't already) and type the following:

!

Make sure that you are using **Python 3** and ***not*** Python 2. On many systems Python 2 is the default, so if you just type "`python`" at a Unix prompt, you will probably get Python 2. To make sure you are using Python 3 type "**`python3`**" at the Unix prompt:

```
$ python3
```

```
>>> import csv
>>> input_file = open('data1.csv', 'r', newline='')
>>> data = csv.reader(input_file)
>>> next(data)
['Adam Smith', '1723', '1790']
>>> del data
>>> input_file.close()
>>> del input_file
```

Of course, normally you would want to read more than just a single line of the file. Usually we will use a `for` loop on the reader object to read each line of the CSV file in turn, as in the script `csv1.py` (shown below (without any comments), and which can be found in the directory of files provided for this topic):

```
import csv
input_file = open('data1.csv', 'r', newline='')
data = csv.reader(input_file)

for line in data:
    [name, birth, death] = line
    print(name, 'was born in', birth, 'and died in', death)
del name, birth, death, line

del data

input_file.close()
```

```
del input_file
```

csv1.py

Exercise 2

(a) Look at the file `data1.csv`. Now run `csv1.py` and observe how it has read each line of `data1.csv`, split it up into fields, and done something with each field.

(b) Look at the file `produce1.csv`. Write a Python script that uses the `csv` module to read each line of the file, and, for each line, print out, on a single line on the screen, the first field, followed by a colon (`:`), followed by the total of all the other fields on that line of the file, e.g. if the first two lines of the file were:

```
apples,12,15,19
oranges,43,29,27
```

then the first two lines your script would print out would be:

```
apples: 46
oranges: 99
```

Writing a CSV file

Now that we know how to read from a CSV file, how do we write to one?

First we open the file (in text mode) for writing (making sure we give `open()` the `newline=''` option). Then we create a special type of object to write to the CSV file. This is the `writer` object, which is defined in the `csv` module, and which we create using the `writer()` function. The `writer` object has a method, the `writerow()` method, that allows us to write a list of fields to the file. Note that the fields can be strings or numbers (or a mixture of both); `writerow()` will convert them if necessary. Also, when using `writerow()` you do not add a new line character (or other EOL indicator) to indicate the end of the line, `writerow()` does it for you as necessary. As information (which you can happily ignore), `writerow()` returns the number of characters it has written to the CSV file. Finally, when we are finished writing to the file, we delete the `writer` object and close the file.

Remember that it is only when the file is closed that the data we've written is committed to the filesystem!

As with the `reader()` function, there are various options we can give the `writer()` function when we create the `writer` object to deal with CSV files that use delimiters other than commas or which quote some (or all) of their fields. We'll meet these options later.

So let's try writing to a CSV file. Type the following in an interactive Python 3 session:

```
>>> import csv
>>> output_file = open('output1.csv', 'w', newline='')
>>> data = csv.writer(output_file)
>>> data.writerow(['H', 'hydrogen', 1, 1.008])
20
>>> data.writerow(['He', 'helium', 2, 4.003])
19
>>> del data
>>> output_file.close()
>>> del output_file
```

Now look at the file you've created by typing the following at the Unix prompt (start a new instance of the Unix shell and change to the directory with the newly created `output1.csv` file if necessary):

```
$ cat output1.csv
H,hydrogen,1,1.008
He,helium,2,4.003
$
```

Of course, normally you would want to write more than just one or two lines to a file. Usually we will use a `for` loop on an iterable to write a whole set of lines to a CSV file, for example in the script `csv2.py` (part of which is shown below, and which can be found in the directory of files provided for this topic):

```
import csv

symbol_to_properties = { … }

output_file = open('atomic_props.csv', 'w', newline='')
data = csv.writer(output_file)

for symbol in symbol_to_properties:
    (name, anum, boil) = symbol_to_properties[symbol]
    data.writerow([symbol, name, anum, boil])
del name, anum, boil, symbol

del data

output_file.close()
del output_file
```

csv2.py

Exercise 3

(a) Open the script `csv2.py` in a text editor and inspect it. Now run the script and examine the file `atomic_props.csv` that it has produced.
(b) Complete the script `exercise3b.py` so that it writes out the contents of the dictionary it contains to a CSV file using the `csv` module.

Delimiters and initial space

Thus far we have only read and written CSV files that use commas as delimiters. What do we do if we want to use a different delimiter (as, for example, in `data2.csv`)?

The `reader()` and `writer()` functions take an optional parameter called `delimiter` which tells the functions what delimiter the CSV file uses, and can be set to any single character string. By default (i.e. if you do not set it) it is set to the comma (`','`), but you can set it to any single character you like. The most common delimiters are the comma (`','`), the TAB (`'\t'`) or a single space (`' '`). To set the delimiter you give the `reader()` or `writer()` function an extra argument (`delimiter=`*`value`*, where *`value`* is a single character string) after the file object associated with the CSV file, e.g.

```
>>> import csv
>>> input_file = open('data2.csv', 'r', newline='')
>>> data = csv.reader(input_file, delimiter='\t')
>>> next(data)
['Adam Smith', '1723', '1790']
```

(If you try the above example, make sure you `del data` and `close()` `input_file` when you are finished.) Compare the above example with what happens if we try and read the `data2.csv` file without specifying that the delimiter is a TAB:

```
>>> import csv
>>> input_file = open('data2.csv', 'r', newline='')
>>> data = csv.reader(input_file)
>>> next(data)
['Adam Smith\t1723\t1790']
```

Because we haven't told the `reader()` function that the delimiter is a TAB, it expects the delimiter to be a comma, and so treats the entire line as a single field (hence returning it to us as a single string). (Again, if you try the above example, make sure you `del data` and `close()` `input_file` when you are finished.)

Now consider the file `data3.csv`. This fields in this file are separated by spaces. (In case you are wondering how the `csv` module knows that "`Adam Smith`" is a single field, rather than two fields, "`Adam`" and "`Smith`", this is because, in `data3.csv`, "`Adam Smith`" appears in quotes (`"Adam Smith"`). We will look at how the `csv` module handles quotes in the next section.) Let's try reading from this file using the `csv` module. We know the fields are separated by spaces so we use `delimiter=' '`:

```
>>> import csv
>>> input_file = open('data3.csv', 'r', newline='')
>>> data = csv.reader(input_file, delimiter=' ')
>>> next(data)
['Adam Smith', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '',
'', '', '', '', '', '', '', '', '', '', '', '', '', '1723', '', '', '', '',
'', '', '1790']
```

(If you try the above example, make sure you `del data` and `close()` `input_file` when you are finished.)

The output from `next()` is not what we might have hoped for. The problem is that in `data3.csv` there are lots of spaces separating the fields (a variable number, in fact). Because we've told it the delimiter is a space, the `csv` module thinks that each space separates two fields, so when it encounters two spaces in succession, it treats this as an empty field and so gives us the empty string.

`data3.csv` is an example of a "**fixed-width**" data file, where each field is of a fixed size, and, if the actual data in the field consists of fewer characters than the chosen size, then the field is padded with blanks. If you print such files on the screen, all the data appears in nicely lined up columns, which looks nice, but they can be a pain to import into some programs. So how do we deal with such files?

For situations like these, there is another optional parameter we can give to the reader() function, called `skipinitialspace`. This is a Boolean value that defaults to `False`. If we set it to `True`, then any "**whitespace**" (spaces, TABs, new line characters, etc.) immediately following the delimiter is ignored. (If it is set to `False` then any whitespace is considered significant and will either be interpreted as part of a field or as another delimiter (if the delimiter is set to a whitespace character).) We can make use of this optional parameter to properly read `data3.csv`.

!

Note that if we are specifying several optional parameters to the `reader()` or `writer()` functions, it doesn't matter what order we give them in, so long as they are listed ***after*** the file object (that we got by opening the CSV file) in the arguments given to the function. So, for example:

```
d = csv.reader(f, delimiter=' ', skipinitialspace=True)
```

and:

```
d = csv.reader(f, skipinitialspace=True, delimiter=' ')
```

are equivalent.

Try the following in an interactive Python 3 session:

```
>>> import csv
>>> input_file = open('data3.csv', 'r', newline='')
>>> data = csv.reader(input_file, delimiter=' ', skipinitialspace=True)
>>> next(data)
['Adam Smith', '1723', '1790']
>>> del data
>>> input_file.close()
>>> del input_file
```

As you can see, that works much better.

Exercise 4

(a) Modify the `exercise4a.py` script so that it gets its data from the file `data3.csv` instead of `data1.csv`. When you've finished, its output should look identical to the output of the `csv1.py` script.

(b) Modify the `exercise4b.py` script so that it gets its data from the file `weird2.csv` instead of `data1.csv` and prints each line of the `weird2.csv` file out in this format:

```
SMITH, Adam: born 1723, died 1790.
```

Quoting

In the previous section, we saw that putting quotes around text had some significance for the `csv` module (e.g. in `data3.csv`). How does this work, and can we change its behaviour if we need to?

Firstly, unlike the way Python in general uses single quotes and double quotes interchangeably to indicate the start and end of a string, the `csv` module only recognises one character as the "quotation" character in a CSV file. By default, this is the double quote character (`"`), but you can change it to be any single character you want. You do this by setting the `quotechar` optional parameter when you call the `reader()` or `writer()` function.

When reading a CSV file, any data surrounded by the character specified in `quotechar` is considered a single field. This is particularly useful for fields which contain the delimiter (e.g. in `data3.csv`) or a new line character (e.g. in `weird4.csv`). When writing a CSV file, any field that contains special characters, such as the delimiter or a new line character, will be "quoted", i.e. the character specified in `quotechar` will be put immediately before and after the field when it is written to the CSV file.

Try the following in an interactive Python 3 session:

```
>>> import csv
>>> output_file = open('output2.csv', 'w', newline='')
>>> data = csv.writer(output_file)
>>> data.writerow(['SMITH, Adam', 1723])
20
>>> del data
>>> output_file.close()
>>>
>>> output_file = open('output3.csv', 'w', newline='')
>>> data = csv.writer(output_file, quotechar="'")
>>> data.writerow(['SMITH, Adam', 1723])
20
>>> del data
>>> output_file.close()
>>> del output_file
```

Now look at the files you've created by typing the following at the Unix prompt (start a new instance of the Unix shell and change to the directory with the newly created `output2.csv` and `output3.csv` files if necessary):

```
$ cat output2.csv
"SMITH, Adam",1723
$ cat output3.csv
'SMITH, Adam',1723
$
```

Note how the field containing "`SMITH, Adam`" is quoted (because it contains a comma, which is the delimiter), and how, in `output3.csv`, single quotes are used for quoting instead of double quotes (because we used `quotechar="'"` when we called the `writer()` function for that file).

You can also control when fields should be quoted when writing a CSV file, and, when reading a CSV file, when quotes around a field should be recognised as quotes (rather than treated as part of the field). This is done using the `quoting` optional parameter of the `reader()` and `writer()` functions. The values that the `quoting` parameter accepts are stored as constants in the `csv` module. These constants, and their meanings, are as follows:

`QUOTE_ALL`	Makes `writer` objects quote *all* fields.
`QUOTE_MINIMAL`	Makes `writer` objects *only* quote those fields which contain *special characters*, such as the delimiter or a new line character. **This is the default.**
`QUOTE_NONNUMERIC`	Makes `writer` objects quote *all non-numeric fields* (i.e. fields whose data cannot be represented as a number). Makes `reader` objects convert all non-quoted fields to floats.
`QUOTE_NONE`	Makes `writer` objects *never* quote fields[3]. Makes `reader` objects do no special processing of quote characters, i.e. any quote characters will be treated as part of the field.

As these constants are defined in the `csv` module, you would normally refer to them by prefixing them with "**`csv.`**".

3 This can cause problems if the field contains the delimiter; for further details on how the `writer` object will try to deal with such situations, see the `QUOTE_NONE` entry in the `csv` module's documentation:
`http://docs.python.org/py3k/library/csv.html#csv.QUOTE_NONE`

Try the following in an interactive Python 3 session:

```
>>> import csv
>>> input_file = open('data2.csv', 'r', newline='')
>>> data = csv.reader(input_file, delimiter='\t', quoting=csv.QUOTE_NONE)
>>> line = next(data)
>>> line
['"Adam Smith"', '1723', '1790']
>>> print(line[0])
"Adam Smith"
```

(Make sure you `del data` and `close()` `input_file` when you are finished.)

As you can see, by setting `quoting` to `QUOTE_NONE` we have caused the double quotes around "Adam Smith" to be considered as part of the field rather than as an indicator that the field contains text.

Now try the following in an interactive Python 3 session:

```
>>> import csv
>>> input_file = open('weird1.csv', 'r', newline='')
>>> data = csv.reader(input_file, quoting=csv.QUOTE_NONNUMERIC)
>>> line = next(data)
>>> line
['SMITH, Adam', 1723.0, 1790.0]
>>> type(line[1])
<class 'float'>
```

(Make sure you `del data` and `close()` `input_file` when you are finished.)

By setting `quoting` to `QUOTE_NONNUMERIC` we have caused the non-text fields to be converted to floats.

And now type the following in an interactive Python 3 session:

```
>>> import csv
>>> output_file = open('output4.csv', 'w', newline='')
>>> data = csv.writer(output_file, quoting=csv.QUOTE_ALL)
>>> data.writerow(['H', 'hydrogen', 1, 1.008])
28
>>> data.writerow(['He', 'helium', 2, 4.003])
27
>>> del data
>>> output_file.close()
>>> del output_file
```

Now look at the file you've created by typing the following at the Unix prompt (start a new instance of the Unix shell and change to the directory with the newly created `output4.csv` file if necessary):

```
$ cat output4.csv
"H","hydrogen","1","1.008"
"He","helium","2","4.003"
$
```

By setting `quoting` to `QUOTE_ALL` we have caused all fields to be quoted regardless of what sort of data they contain.

Exercise 5

(a) Modify the `exercise5a.py` script so that it gets its data from the file `weird6.csv` instead of `data1.csv`. When you've finished, its output should look identical to the output of the `csv1.py` script.

(b) Complete the script `exercise5b.py` so that it writes out the contents of the dictionary it contains to a CSV file using the `csv` module. In the CSV file the script creates, fields should be separated from each other by a TAB, and the data contained in each field should be preceded and followed by an asterisk (`*`), regardless of the type of data in the field.

Dialects

As we have seen, there are a number of options[4] we can give to the reader() and writer() functions to change how they handle the CSV file. Whilst using these frequently will help you remember them, it is also the case that if you have to use the same set of options frequently then it will get tedious having to specify them every time. (And if you don't use them frequently then you'll probably have to look them up each time you do come to use the `csv` module, which can also be irritating.) Wouldn't it be great if there was a single setting you could use for the most common sets of options?

The `csv` module has a number of built-in collections of settings for reading common types of CSV files. These are called "**dialects**" and you specify them by using the `dialect` optional parameter of the `reader()` and `writer()` functions. The two most useful dialects are:

`excel`	This sets the various options to the appropriate values for the most usual sort of Excel-generated CSV file. **This is the default.**
`excel-tab`	This sets the various options to the appropriate values for the most usual sort of Excel-generated TAB-delimited text file.

In Python 3.2 (but not earlier versions of Python 3) another built-in dialect was added:

`unix`	This sets the various options to the appropriate values for the most usual sort of CSV file generated on Unix systems. In particular[5], it causes all fields in the CSV file to be quoted.

You can also create your own dialects, but that's beyond the scope of these notes; consult the `csv` module's documentation for further information about doing this:

`http://docs.python.org/py3k/library/csv.html`

Let's try out one of these dialects. `data2.csv` is a TAB-delimited file created from a spreadsheet program, so it would make sense to try the `excel-tab` dialect. For comparison purposes we'll try first reading the file without explicitly using any of the optional parameters. Type the following in an interactive Python 3 session:

4 And there are even more that we haven't covered here because they aren't used very often. For details of all the options that the `reader()` and `writer()` functions accept, see the `csv` module's documentation:
`http://docs.python.org/py3k/library/csv.html#dialects-and-formatting-parameters`

5 It also explicitly sets the EOL indicator to be the new line character ('`\n`'), which is not something we usually need to worry about (which is why we haven't covered how to explicitly set the EOL indicator in these notes).

```
>>> import csv
>>> input_file = open('data2.csv', 'r', newline='')
>>> data = csv.reader(input_file)
>>> next(data)
['Adam Smith\t1723\t1790']
>>> del data
>>> input_file.close()
>>> del input_file
>>>
>>> input_file = open('data2.csv', 'r', newline='')
>>> data = csv.reader(input_file, dialect='excel-tab')
>>> next(data)
['Adam Smith', '1723', '1790']
```

(Make sure you `del data` and `close()` `input_file` when you are finished.)

As you can see, the `excel-tab` dialect works (and saves us having to remember what options we need to set for this sort of file). Hurrah!

Final exercise

And we end with an exercise:

Exercise 6

(a) Look at the file `produce2.csv`. Write a Python script that uses the `csv` module to read each line of the file, and, for each line, print out, on a single line on the screen, the first field, followed by a colon (`:`), followed by the second field multiplied by the total of all the other fields on that line of the file, e.g. if the first two lines of the file were:

```
"apples"     0.5   12   15   19
"oranges"    0.7   43   29   27
```

then the first two lines your script would print out would be:

```
apples:     23.0
oranges:    69.3
```

(b) Modify the script you wrote in the previous part of this exercise, so that, as well as printing to the screen, it also writes to a file named `produce2.out` using the `csv` module. Each line of this file should contain three fields:

the first field from `produce2.csv`,
the total you calculated for that field, and
the total multiplied by the second field from `produce2.csv`.

The fields of `produce2.out` should be separated by TABs and each field should be surrounded by single quotes.
So, if the first two lines of the `produce2.csv` were:

```
"apples"     0.5   12   15   19
"oranges"    0.7   43   29   27
```

then the first two lines of `produce2.out` would be:

```
'apples'    '46'  '23.0'
'oranges'   '99'  '69.3'
```