Handling files

Duration: 90 minutes

Learning Objectives:

- Be able to read and write to CSV files

Modelling our data

[Download CSV file from kaggle.com, or from the link in the email (need to rename file extension from .txt to .csv)]

> You need to register to kaggle.com - it's free, you can use your Facebook/Google account to sign in!

Data visualisation starts with data analysis - and analysis starts with a dataset. Our current dataset is going to be the top 50 bestselling Amazon books from 2009-2019, courtesy of Amazon. The techniques shown here, however, can be used with any dataset, even different formats, with a bit of research.

We are going to focus on .csv data, which stands for Comma Separated Values - each row represents a new entity, while each column is a different attribute of an entity, with the first row showing the column names usually.

> Create a new file called `reading_from_csv.py`

In order to do anything meaningful, we have to turn our .csv file into meaningful data.

There are many ways to execute this, you can use custom objects or named tuples if you want to, but we will show you one of the most basic solutions that is applicable across all languages - we will create a list of dictionaries, just like the ones we have been working with up to this point!

Processing the data

We will import a built-in package called csv. There are some packages that, even though they need to be imported, don't need to be installed with pip - they exist in the core python language, but we still need to import them.

We will set up an initially empty list to hold the books we get from the csv.

Sometimes CSV files can be produced with white space at the start of the lines (normally if people use spaces after the commas) so we can add a flag here to skip the initial space in the line.

```
import csv
books = []

with open("bestsellers with categories.csv", "r") as csvfile:
    print(csvfile)
```

There are at least three interesting things happening here:

- You might not have seen the with statement before; this makes sure that the file is closed again, once we're finished with it. Otherwise, we would have to manually close it.
- `open` has a number of default parameters set, which controls how it behaves.
- The `r` that we're passing in denotes that we want to open this file in read mode - this will happen by default whether we pass the r in or not, but it can be good to be explicit about what our intentions are. Opening it in read mode means we can't amend the file at the moment, but this is fine for our purposes!
- Finally, we've now got a reference to the file we've opened in a variable called `csvfile`.

Once printed out, you can see that it is an interesting data type, but currently it is not useful for us.

Reading data

Let's continue. Our next job is to set up a `csv.reader` object. This will let us loop through each line of the CSV file. We use the `csv.reader()` function, that comes with the CSV library, and we pass in the file that we opened.

```
import csv
books = []

with open("bestsellers with categories.csv", "r") as csvfile:
    print(csvfile)
    reader = csv.reader(csvfile, skipinitialspace=True)
```

`skipinitialspace=True` makes sure that the spaces after our delimiters will be ignored if they exist. Not mandatory, but good to include/remember!

Great! Now that we're all set up, we can begin to loop through our reader object. We would be able to find this out by reading the relevant docs!

```
import csv
books = []

with open("bestsellers with categories.csv", "r") as csvfile:
    print(csvfile)
    reader = csv.reader(csvfile, skipinitialspace=True)

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

Here, we're printing out the individual rows.

It looks like each row is a list. Great - we know how to work with that! At this point, we could actually complete our task just using `row[0], row[1]` etc. But let's make our code more readable by creating `dict`s that will represent books!

```
import csv
books = []

with open("bestsellers with categories.csv", "r") as csvfile:
    print(csvfile)
    reader = csv.reader(csvfile, skipinitialspace=True)

    for row in reader:
        new_book = {}
        new_book["name"] = row[0]
        new_book["author"] = row[1]
        new_book["user_rating"] = row[2]
        new_book["reviews"] = row[3]
        new_book["price"] = row[4]
        new_book["year"] = row[5]
        new_book["genre"] = row[6]
        books.append(new_book)
print(books)
```

And lastly print out each book's details.

We have a problem however. Let's check out the first entry!

The first row is the column titles, and as such, we do not consider this a book.

There's a simple solution for this problem; we can just call the `next()` function on our `reader` instance to "fast-forward" past the first row before we start iterating!

```
# Added
next(reader)
for row in reader:
```

Now if we run the program we should see the results printed out.

However, we still have quite a bit of ugliness in our program. These lines, in particular, won't scale very well:

```
    for row in reader:
        new_book = {}
        new_book["name"] = row[0]
        new_book["author"] = row[1]
        new_book["user_rating"] = row[2]
        new_book["reviews"] = row[3]
        new_book["price"] = row[4]
        new_book["year"] = row[5]
        new_book["genre"] = row[6]
        books.append(new_book)
```

What if we had a hundred columns in our CSV file? Would we have to type all the way to row[n]?

Let's explore other options!

Named Tuples

Named tuples are fantastic in Python when it comes to representing more complex data. Although dicts can be used, but they can get clunky easily. Let's see what namedtuples can do for us!

First of all, they need to be imported like the csv package. Once imported, we can create a new named tuple that we can name ourselves - and we can also give it a list of attributes in a specific order, so it can allocate those values to specific attrbiutes. Kind of like how we can create dicts, but this way, we can simplify our code considerably.

```
# At the top
from collections import namedtuple

Book = namedtuple("Book", "name author user_rating reviews price year ge
```

The first argument is the name of the namedtuple, it's mostly for the internal workings of

Python, but it's not majorly important. The second argument however needs to be a string where the whitespace separation creates new attributes for us - instead of us saying new_book["name"], we can just write the value once we create a new Book!

After writing the above, in order to create a new book, we could do something like this:

```
new_book = Book("Lord of the Rings", "J.R.R.Tolkien", ...)
```

It would allocate each argument to the corresponding value, so "Lord of the Rings" would be saved under the attribute "name".

The other convenient feature is that we can access each key-value pair by using the `.` notation instead of `[]`, like so:

```
print(new_book.name)
```

Let's rewrite our entire code to accomodate to this!

```
import csv
from collections import namedtuple
Book = namedtuple("Book", "name author user_rating reviews price year ge
books = []

with open("bestsellers with categories.csv", "r") as csvfile:
    reader = csv.reader(csvfile, skipinitialspace=True)
    next(reader)

    for row in reader:
        new_book = Book(row[0], row[1], row[2], row[3], row[4], row[5],
        books.append(new_book)
print(books)
```

Already a huge improvement, but we can make it even better!

We can use the * operator before lists to break it down into individual elements and pass them as arguments - so it doesn't matter how many columns there are in future .csv files, as long as we add the columns as attributes in the namedtuple creator, we can scale it infinitely!

```
new_book = Book(*row)
```

Wonderful!

Remember - we can probably still refactor this. One of the coolest features of Python is using something called List Comprehensions!

List comprehensions enable us to simplify writing code where the end result is a list. We can use it when we are creating our list of books. You can also add boolean values at the end of a list comprehension in an if statement to only return items into a new list conditionally!

> Note that this is completely optional - if you don't feel comfortable with list comprehensions, leave the code as is! There is no functional difference, only the amount of code needed to be written is reduced!

```
import csv
from collections import namedtuple
Book = namedtuple("Book", "name author user_rating reviews price year ge

with open("bestsellers with categories.csv", "r") as csvfile:
    reader = csv.reader(csvfile, skipinitialspace=True)
    next(reader)

    books = [Book(*row) for row in reader]

print(books)
```

Mini-lab

Let's practice looping through and processing the data.

- Create a list with all of the bestsellers from 2009 to 2012!
- How expensive is the most expensive book?
- Create a list with all books whose author has the first name George!

`Hint 1` : You can use either comprehensions or loops for these exercises. Don't forget you can add if statements to the end of the list comprehensions!

`Hint 2` : You probably have to change certain datatypes - remember `int()` ?

`Hint 3` : Remember you can use the `if substring in string:` syntax to check if a substring can be found in another string!

Solutions

Create a list with all of the bestsellers from 2009 to 2012:

```
filtered_bestsellers = []

for book in books:
    if(int(book.year) >= 2009 and int(book.year) <= 2012):
        filtered_bestsellers.append(book)
print(filtered_bestsellers)
```

How expensive is the most expensive book?

```
# ...
most_expensive = books[0]
for book in books:
    if int(book.price) > int(most_expensive.price):
        most_expensive = book
print(most_expensive)
```

Create a list with all books whose author has the first name George!

```
# ...
george_books = []
for book in books:
    if("George" in book.author):
        george_books.append(book)
print(george_books)
```

Writing to csv

Now that we've got a list of book objects, we can open the file for writing, amend the objects as we need to, and write them to a file.

Let's say we want to create a modified bestsellers list to display prices in £ instead of $!

First we need to create another list containing the modified data. At the time of writing, $1 equals £0.79.

```
import csv
from collections import namedtuple
Book = namedtuple("Book", "name author user_rating reviews price year ge

with open("bestsellers with categories.csv", "r") as csvfile:
    reader = csv.reader(csvfile, skipinitialspace=True)
    next(reader)

    books = [Book(*row) for row in reader]
    modified_books = []
    for book in books:
        modified_books.append(Book(book.name, book.author, book.user_rat
```

Next, we need to create a file, which we can do with another `with open()` command, but this time in write mode! Be aware - every time you open a file in write mode, it will destroy the contents of the file, so make sure to take a copy of the contents in your Python code! We are also setting `newline=''` - when we are writing or appending to csv files, in certain versions (mainly on Windows) a new line might be added, which, due to it being a newline character, could break our reading after the first time running the script.

```
with open("bestsellers_british_price.csv", "a", newline='') as csvfile:
    writer = csv.writer(csvfile, quoting=csv.QUOTE_ALL)
```

This looks much the same as creating a reader object, but this time we also pass in an additional keyword argument: quoting. This ensures that each of the columns have their values properly quoted.

Finally, we can append to the file by calling `writer.writerow(data):`, which can be a named tuple!

```
with open("bestsellers_british_price.csv", "a", newline='') as csvfile:
    writer = csv.writer(csvfile, quoting=csv.QUOTE_ALL)
    for book in modified_books:
        writer.writerow(book)
```

This should create a brand new csv file right next to our script!

Conclusion

In this lesson, you have seen how to construct namedtuples, call methods on them, and use them to interact with an external resource. (A CSV file.)

We've seen how to work with files, and use objects, list comprehensions, and the * operator in a small program.