CSV

Introduction ¶

In this lesson we will cover the CSV (comma-separated values) data format, one of the most simple and popular data serialization formats used in data science.

Objectives

You will be able to:

- Describe features of the CSV format and the Python `csv` module
- Use Python to read and parse data from CSV files
- Interpret the schemas of CSV files

CSV Format

The full name for this data format is "comma-separated values", but it is conventionally referred to by the shorthand "CSV"

At a high level, this format is designed to contain tabular data (data represented by rows and columns), where each line of the file is a row, and each column of data within that row is separated by a comma.

CSV is a **plain text**, **delimited** format, meaning its contents are essentially just strings with a specified structure. You can visually inspect the contents of plain text files using general-purpose text editing tools such as Vim, VS Code, or Notepad. (This is in contrast to formats like images or SQLite databases, which contain encoded bytes that require specific tools to read.)

Because of this, we can start digging in to how the CSV *format* works without actually using any CSV *files*. Instead we'll just start with some Python strings.

Let's say we have this dataset representing the 100 meter dash times for 4 high school athletes across 3 track meets:

Meet 1	Meet 2	Meet 3
13.10	13.59	13.44
13.93	13.85	13.47
14.12	14.41	13.89
14.42	13.55	13.43

If we wanted to represent that dataset in Python, we might do something like this:

In [28]:
```
track_times = [
    [13.10, 13.59, 13.44],
    [13.93, 13.85, 13.47],
    [14.12, 14.41, 13.89],
    [14.42, 13.55, 13.43]
]
track_times
```

Out[28]:
```
[[13.1, 13.59, 13.44],
 [13.93, 13.85, 13.47],
 [14.12, 14.41, 13.89],
 [14.42, 13.55, 13.43]]
```

But as discussed previously, that data is currently loaded *in memory* using Python, but we often want to be able to store things *on disk* using some kind of file.

So first, we need to *serialize* the data in some way. Serialization means that we are converting the Python object into a structured format for storage and sharing. Let's serialize this list of lists so that:

- the overall list is represented by a string
- each nested list is on its own line of the string (i.e. separated by a newline)
- each element within each nested list is separated by a comma

In [29]:
```
# Initialize an empty string
track_times_csv = ""

# Loop over all lists in the overall list
for index, athlete_times in enumerate(track_times):
    # Join together the values in the nested list using
    # a comma as a separator
    athlete_times_string = ",".join([str(time) for time in athlete_times])
    # Append the values to the overall string
    track_times_csv += athlete_times_string
    # Append a newline, unless this is the last row
    if index < (len(track_times) - 1):
        track_times_csv += "\n"

print(track_times_csv)
```

```
13.1,13.59,13.44
13.93,13.85,13.47
14.12,14.41,13.89
14.42,13.55,13.43
```

...and that's it! That's a CSV.

We can write it to a file like this:

In [30]:
```
with open("track_times.csv", "w") as f:
    f.write(track_times_csv)
```

Then later even if we restart the kernel, we can open up the file and deserialize the data back into a list of lists:

In [31]:
```
with open("track_times.csv") as f:
    track_times_csv_from_disk = f.read()
print(track_times_csv_from_disk)
```

```
13.1,13.59,13.44
13.93,13.85,13.47
14.12,14.41,13.89
14.42,13.55,13.43
```

In [32]:
```
track_times_from_disk = []
for row in track_times_csv_from_disk.split("\n"):
    times = [float(time) for time in row.split(",")]
    track_times_from_disk.append(times)

track_times_from_disk
```

Out[32]:
```
[[13.1, 13.59, 13.44],
 [13.93, 13.85, 13.47],
 [14.12, 14.41, 13.89],
 [14.42, 13.55, 13.43]]
```

If we did everything correctly, this new list of lists should contain the exact same data as the original:

In [14]:
```
track_times_from_disk == track_times
```

Out[14]:
```
True
```

Great!

`csv` Module

That wasn't too bad, but in general you don't actually need to serialize and deserialize your data "by hand" like that. Instead, there is a module built in to Python called `csv` that can do the same thing using fewer lines of code! It also has helpful functionality for dealing with:

- CSV files with headers that indicate what each column represents
- More kinds of plain text delimited data, such as tab-separated values (TSV) files (where the delimiter is a tab `\t` rather than a comma)
- Properly handling text data inside a CSV, e.g. if your data contains the text `"Hello, World!"` you want to make sure that the `,` is treated as part of the contents of that cell, not treated as a delimiter separating the columns
- Reading and writing CSV files that are compatible with spreadsheet software such as Excel

You can find full documentation for this module here (https://docs.python.org/3/library/csv.html).

To use the `csv` module, start by importing it:

In [15]:
```
import csv
```

csv.reader

If we wanted to replicate the previous example of opening the track times CSV, this time using the `csv` module, that would look like this:

In [17]:
```
with open("track_times.csv") as f:
    # Pass the file in to a "reader" object and specify that
    # values without explicit quotes (i.e. all values in this
    # dataset) should be treated as numbers
    reader = csv.reader(f, quoting=csv.QUOTE_NONNUMERIC)
    # Get all of the data from the reader using `list`
    track_times_with_csv_module = list(reader)

track_times_with_csv_module
```

Out[17]:
```
[[13.1, 13.59, 13.44],
 [13.93, 13.85, 13.47],
 [14.12, 14.41, 13.89],
 [14.42, 13.55, 13.43]]
```

We shortened the code reading the data from a file into a list of lists from 6 lines to 3, and all of it is quite simple (no joining, splitting, or list comprehensions). We could even shrink that to two lines (opening the file, then nesting `list(csv.reader(f, quoting=csv.QUOTE_NONNUMERIC))`). Nice!

(Note: calling `list()` on the reader is important if you want to be able to use the data after the file is closed. Like we discussed previously relating to opening files in general, sometimes the tools do not assume that you want to read all of the file contents into memory at once, so you have to do this explicitly. If you do not call `list()`, the reader will allow you to read through the file one line at a time and will throw an error if the file is closed.)

What we just used is the `csv.reader` object (documentation here (https://docs.python.org/3/library/csv.html#csv.reader)), which is useful for CSVs without column headings like the one we created. It has a parallel `csv.writer` object (documentation here (https://docs.python.org/3/library/csv.html#csv.writer)) that can take a list of lists and write it to a CSV file with similarly few lines of code.

However most of the time you will be working with CSVs that have column headings, in which case the `csv.DictReader` is often more useful.

csv.DictReader

Let's advance from a high school track dataset to a dataset with information about Olympic track and field medal-winners from kaggle (https://www.kaggle.com/jayrav13/olympic-track-field-results). The first five rows look like this:

Gender	Event	Location	Year	Medal	Name	Nationality	Result

Gender	Event	Location	Year	Medal	Name	Nationality	Result
M	10000M Men	Rio	2016	G	Mohamed FARAH	GBR	25:05.17
M	10000M Men	Rio	2016	S	Paul Kipngetich TANUI	KEN	27:05.64
M	10000M Men	Rio	2016	B	Tamirat TOLA	ETH	27:06.26
M	10000M Men	Beijing	2008	G	Kenenisa BEKELE	ETH	27:01.17
M	10000M Men	Beijing	2008	S	Sileshi SIHINE	ETH	27:02.77

Technically we could open this with `csv.reader`:

In [21]:
```
with open("olympic_medals.csv") as f:
    reader = csv.reader(f)
    # Printing only the header and first 5 rows of data
    for _ in range(6):
        print(next(reader))
```

```
['Gender', 'Event', 'Location', 'Year', 'Medal', 'Name', 'Nationality', 'Resul
t']
['M', '10000M Men', 'Rio', '2016', 'G', 'Mohamed FARAH', 'GBR', '25:05.17']
['M', '10000M Men', 'Rio', '2016', 'S', 'Paul Kipngetich TANUI', 'KEN', '27:05.
64']
['M', '10000M Men', 'Rio', '2016', 'B', 'Tamirat TOLA', 'ETH', '27:06.26']
['M', '10000M Men', 'Beijing', '2008', 'G', 'Kenenisa BEKELE', 'ETH', '27:01.1
7']
['M', '10000M Men', 'Beijing', '2008', 'S', 'Sileshi SIHINE', 'ETH', '27:02.7
7']
```

Then we could use list indexing to access the gender field of a given row using `[0]` or the nationality field using `[-2]`. But you can see how that would create some hard-to-read and error-prone code!

Fortunately we aren't limited to just using the `list` data structure — we can use a `dict` instead, so that we could look up the gender field using `["Gender"]` or the nationality field using `["Nationality"]`.

In order to read the data in as a list of dictionaries rather than a list of lists, we can use `csv.DictReader`:

In [ ]:
```
```

In [33]:
```
with open("olympic_medals.csv") as f:
    reader = csv.DictReader(f)
    olympics_data = list(reader)

# Print the first 5 rows of data
for index in range(5):
    print(olympics_data[index])
```

```
{'Gender': 'M', 'Event': '10000M Men', 'Location': 'Rio', 'Year': '2016', 'Meda
l': 'G', 'Name': 'Mohamed FARAH', 'Nationality': 'GBR', 'Result': '25:05.17'}
{'Gender': 'M', 'Event': '10000M Men', 'Location': 'Rio', 'Year': '2016', 'Meda
l': 'S', 'Name': 'Paul Kipngetich TANUI', 'Nationality': 'KEN', 'Result': '27:0
5.64'}
{'Gender': 'M', 'Event': '10000M Men', 'Location': 'Rio', 'Year': '2016', 'Meda
l': 'B', 'Name': 'Tamirat TOLA', 'Nationality': 'ETH', 'Result': '27:06.26'}
{'Gender': 'M', 'Event': '10000M Men', 'Location': 'Beijing', 'Year': '2008',
'Medal': 'G', 'Name': 'Kenenisa BEKELE', 'Nationality': 'ETH', 'Result': '27:0
1.17'}
{'Gender': 'M', 'Event': '10000M Men', 'Location': 'Beijing', 'Year': '2008',
'Medal': 'S', 'Name': 'Sileshi SIHINE', 'Nationality': 'ETH', 'Result': '27:02.
77'}
```

The number of rows in the dataset is just the length of the resulting list:

In [11]:
```
len(olympics_data)
```

Out[11]:
```
2394
```

Now we can perform data analysis and cleaning tasks in a neater, clearer way.

For example, if our task was ***filter the data so that it only includes gold medals***, that logic would look something like this:

In [12]:
```
gold_medals = []

for row in olympics_data:
    if row["Medal"] == "G":
        gold_medals.append(row)

print(f"""Out of {len(olympics_data)} total medals, this dataset
contains information about {len(gold_medals)} gold medals""")
```

```
Out of 2394 total medals, this dataset
contains information about 799 gold medals
```

Or if it was ***for all USA gold medals in 2016, what were the events and the names of the athletes***, that logic would look something like this:

In [13]:
```
usa_2016_gold_medals = []

for row in olympics_data:
    if row["Medal"] == "G" and row["Nationality"] == "USA" and row["Year"] == "20
        usa_2016_gold_medals.append({"Event": row["Event"], "Name": row["Name"]})

usa_2016_gold_medals
```

Out[13]:
```
[{'Event': '1500M Men', 'Name': 'Matthew CENTROWITZ'},
 {'Event': '400M Hurdles Men', 'Name': 'Kerron CLEMENT'},
 {'Event': '4X400M Relay Men', 'Name': 'null'},
 {'Event': 'Decathlon Men', 'Name': 'Ashton EATON'},
 {'Event': 'Long Jump Men', 'Name': 'Jeff HENDERSON'},
 {'Event': 'Shot Put Men', 'Name': 'Ryan CROUSER'},
 {'Event': 'Triple Jump Men', 'Name': 'Christian TAYLOR'},
 {'Event': '100M Hurdles Women', 'Name': 'Brianna ROLLINS'},
 {'Event': '400M Hurdles Women', 'Name': 'Dalilah MUHAMMAD'},
 {'Event': '4X100M Relay Women', 'Name': 'null'},
 {'Event': '4X400M Relay Women', 'Name': 'null'},
 {'Event': 'Long Jump Women', 'Name': 'Tianna BARTOLETTA'},
 {'Event': 'Shot Put Women', 'Name': 'Michelle CARTER'}]
```

And we could write that result to a file using `csv.DictWriter` (documentation here (https://docs.python.org/3/library/csv.html#csv.DictWriter)):

In [14]:
```
with open("usa_2016_gold_medals.csv", "w") as f:
    writer = csv.DictWriter(f, fieldnames=["Event", "Name"])
    writer.writeheader()
    for row in usa_2016_gold_medals:
        writer.writerow(row)
```

We can use the bash command `cat` to visually inspect the file that was created:

(If you are running this lesson on a Windows or other non-Unix computer, you may need to use some other tool to do this, or just skip this step since it is just for demonstration purposes.)

In [15]:
```
! cat usa_2016_gold_medals.csv
```

```
Event,Name
1500M Men,Matthew CENTROWITZ
400M Hurdles Men,Kerron CLEMENT
4X400M Relay Men,null
Decathlon Men,Ashton EATON
Long Jump Men,Jeff HENDERSON
Shot Put Men,Ryan CROUSER
Triple Jump Men,Christian TAYLOR
100M Hurdles Women,Brianna ROLLINS
400M Hurdles Women,Dalilah MUHAMMAD
4X100M Relay Women,null
4X400M Relay Women,null
Long Jump Women,Tianna BARTOLETTA
Shot Put Women,Michelle CARTER
```

Data Schemas and CSV Files

A data schema is, broadly, the structure that the data is stored in. As a part of the overall data science process, understanding the schema is an important element of **data understanding**.

It includes some things that can be determined automatically/programmatically:

- How many rows and columns are there?
- What are the column names?
- For a given column, what is the data type (e.g. string, integer, boolean)?
- Is one of the columns a unique identifier for that record?
- Which columns can contain missing data?

Then there are other things that you might need additional documentation to determine:

- What does a row (record) represent?
- What does each column (feature) represent?
- For a numeric column, what are the units (e.g. seconds, inches, kilograms)?

When working with database tools like SQL, the schema is a formal attribute of the system that can be inspected. In the SQLite command-line interface, for example, the command is `.schema`, which will tell you the names of all tables and the data types of each of their columns. There can also be formal schemas in NoSQL (not only SQL) databases as well as JSON and XML file formats.

When working with CSV files, understanding the schema is more ambiguous and open-ended. It's about getting yourself oriented to the overall structure of the data, and filling in the finer details as needed for the business problem.

Data Schema of the Olympic Medals Dataset

Let's take the Olympic medals dataset we've been working with as an example.

Possibly the most important question to ask about any CSV dataset is: *What does a row represent?*

This might seem extremely obvious but it can be important to make sure you understand the details.

With this dataset, a row represents an Olympic medal win.

It does *not* represent:

- A country (the same country can appear multiple times if it has won medals in multiple years)
- A type of Olympic track and field event (the same event name can appear multiple times if it happened in multiple years)
- An Olympic athlete (the same athlete can appear multiple times if they have won multiple medals, and some rows are team events where the name is `null`)
- The performance ("Result") of a given athlete in a given event in a given year, if that performance did not win a medal
- etc.

Therefore if you were trying to answer a question about a country, a type of event, or an athlete, you would need to perform some additional filtering and/or aggregation in order to form the correct units to answer that question.

If you were trying to answer a question about the performance of athletes who did not win medals vs. those who did win medals, that would not be possible with this dataset, since this dataset only includes medal winners. It also would not allow you to answer questions about the performance of these athletes at these events outside of the Olympics (e.g. at the national championship level).

Understanding what a row represents is crucial for being able to identify what questions you can and cannot answer with a given dataset.

What does each column represent?

To extract the basics of this information, you can just call the `.keys()` on one of the row dictionaries. In this case, the columns are:

- `'Gender'`
- `'Event'`
- `'Location'`
- `'Year'`
- `'Medal'`
- `'Name'`
- `'Nationality'`
- `'Result'`

If you need to understand further details such as the capitalization style of the names or the units of the results, you can often find this information in documentation about the dataset. If not, you may need to apply your own judgment.

For example, let's look at a record from this dataset:

In [16]:
```
olymp ics_data[85]
```

Out[16]:
```
{'Gender': 'M',
 'Event': '100M Men',
 'Location': 'Montreal',
 'Year': '1976',
 'Medal': 'S',
 'Name': 'Donald QUARRIE',
 'Nationality': 'JAM',
 'Result': '10.08'}
```

What do we think are the units of the `Result` value?

Well, if we look at the `Event` value, it is a 100 meter dash. That is a track event, so these are some kind of units of *time*.

But are they hours? Minutes? Seconds?

At this point if you are unsure, you could look up recent 100 meter dash times, or the world record best time for that event. (Although you'll want to watch out for older data like this — sometimes data from the '70s is going to have a significantly different scale compared to data from today!)

With a bit of research, it's clear that the units are *seconds*.

But what if we look at a different record:

In [17]:
```
olympics_data[1100]
```

Out[17]:
```
{'Gender': 'M',
 'Event': 'Discus Throw Men',
 'Location': 'Los Angeles',
 'Year': '1984',
 'Medal': 'B',
 'Name': 'John POWELL',
 'Nationality': 'USA',
 'Result': '65.46'}
```

Is that `Result` also measured in seconds?

No, this is a discus throw, which is a field event. The goal is not to finish in the least amount of time, it's to throw the discus as far as possible. So this is a unit of *distance*.

If we look up some more information about this event, we find that the units are *meters*.

It appears that the `Result` column contains different units depending on the event, which makes sense, since the result needed to win is different depending on the event.

As you can see, this process can take a fair amount of time if you do not have domain knowledge related to the dataset. Sometimes it is a good strategy to learn more about the columns one at a time — only as you encounter business questions that seem related to them — rather than trying to understand all of the columns before performing any analysis.

Is there a unique identifier?

In some datasets, one of the columns is a unique identifier, also commonly shorted as just "ID" or "id".

For example, if each record represented a country, there might be a unique identifier like `USA`, `KEN`, etc. Or a zip code for a geographical area or a social security number for a person. These unique identifiers are genuine features of the rows, and they can be helpful for merging datasets together by connecting a feature of one dataset to an identifier of another.

In other cases, there are unique identifiers that are solely artifacts of the particular dataset and are not useful for merging. For example, if we went through all 2394 records in the Olympic medals dataset and assigned them IDs 0 through 2393, we would have a unique identifier, but it would not be a genuine feature of a given row and would not be useful for merging together additional data. SQL databases often have this type of unique identifier, called a "primary key".

This dataset does not have a unique identifier, since almost all of the values appear more than once. There are multiple gold medals, multiple events from 2016, multiple men's events, etc. Nevertheless we could potentially merge it with other tables if *those* tables had one or more of these features. For example, we could add in the GDP of the athlete's home country for that year if we found a table of countries labeled similarly and their GDP by year.

Schemas Conclusion

Now that you have the tools to start working with actual datasets, your tasks as a data scientist have expanded beyond just technical code implementation tasks into analysis tasks. Often there will not be a single "right answer" and you will need to contend with ambiguity and uncertainty about what is in your dataset and what questions you can answer.

Considering the schema is a great place to start when answering these questions!

One way to get more practice with this is open-ended data exploration. For example, check out this website (https://dataportals.org/search) for a list of open data portals, many of which offer CSV data. You can even explore the collection of data portals programmatically using a CSV (https://raw.githubusercontent.com/okfn/dataportals.org/master/data/portals.csv)!

Summary

In this lesson, you learned the basics of the CSV data format and looked at an example using pure Python to serialize data. Then you were introduced to the simpler, more sophisticated technique of using the `csv` module. Finally, you were introduced to some of the concepts of data schemas in general, how they apply to CSV formats, and what questions to ask when you encounter a new dataset.