

COM6018 Data Science with Python

Week 2 - Reading and Writing Data Files

Varvara Papazoglou

Overview

- What is a dataset?
- Data formats used in Data Science
 - CSV and TSV files
 - JSON files
 - Other formats
- Good practice in dataset creation
- APIs

Datasets vs Databases

Data scientists often talk about `dataset` ; this should not be confused with a `database` .

- A `dataset` is a structured collection of data, typically stored in a *single file*.
- A `database` is a broader concept, typically a collection of datasets, stored in a database management system (DBMS).

Used for different purposes:

- A database will typically be designed to allow *efficient querying*, i.e. recalling information, cross-referencing items, etc.
- Data scientists are more often interested in *learning* something by using the whole dataset.

e.g. compare 'what was the weather yesterday?' with 'what will the weather be tomorrow?'

Structure of a dataset

All datasets have a common structure:

- A dataset is a collection of records (or data items)
- Each record is a collection of fields (also called attributes or features)

For example, consider a spreadsheet of student data:

- Each row represents a student (a single record)
- For every row, there are the same set of columns (fields), e.g., name, degree title, date of birth, etc.

Note, the fields often have simple types (e.g., a string) but could have more complex types (e.g., a list or a set).

Using a dataset

- The data we use will typically be provided to us as a dataset stored in a file (or files).
 - Note though, that the data may originally have come from some sort of sensor, or from human input, or from a larger database.
- We will need to read the data from the file into our programs.
- We will then need to process the data in some way, e.g., to extract information, to find patterns, to make predictions, etc.
- We may then need to write the results of our processing back to a file.
 - Perhaps as a new dataset that will be used by another program or another group of people.
- So, understanding how to load and save datasets is a key skill for a data scientist.

Human-readable vs Machine-readable

Human-readable formats are designed to be read by humans

- Typically text based, e.g., ASCII or Unicode text files.
- Can be loaded into a text editor or printed to a terminal.
- Examples: CSV, TSV, JSON, XML, YAML
- Suitable for small or medium sized datasets.
- Downside: not very compact, not very fast to load.

Human-readable vs Machine-readable

Machine-readable formats are designed to be read by computers

- Typically binary files, i.e. not text files, close to the internal representation of the data.
- Often very compact, and fast to load.
- Examples: Avro, HDF5, Parquet
- Suitable for large datasets.
- Downside: not human readable, so hard to check/debug.

Data Formats used in COM6018

In this module we will be primarily using two human-readable formats:

- CSV (Comma Separated Values)
- JSON (JavaScript Object Notation)

We may also encounter a few other formats for specific tasks, e.g. YAML for configuration files, specific binary formats for storing image data, etc.

CSV (Comma Separated Values)

A simple text-based format for storing tabular data.

- Each record is stored on a separate line.
- Each field is separated by a comma.
- The first line may contain the names of the fields.

Example, a CSV file containing information about wind farms:

```
"id", "turbines", "height", "power"  
"WF1355", 13, 53, 19500  
"WF1364", 3, 60, 8250  
"WF1356", 12, 60, 24000  
"WF1357", 36, 60, 72000
```

TSV (Tab Separated Values) is similar, but uses tabs instead of commas.

Reading with csv.reader

CSV files can be read with the `reader` function from the `csv` module (included in the Python Standard Library).

```
import csv

# Open the file
with open('data/020/windfarm.csv') as csvfile:

    # Attach a reader to the file
    windfarm_reader = csv.reader(csvfile, skipinitialspace=True)

    # Read each row (as a list) and store them as a list of lists.
    data = [row for row in windfarm_reader]
```

Reading with csv.reader

The code on the previous slide will return a list of lists

```
[  
  ['id', 'turbines', 'height', 'power'],  
  ['WF1355', '13', '53', '19500.0'],  
  ['WF1364', '3', '60', '8250.0'],  
  ['WF1356', '12', '60', '24000.0'],  
  ['WF1357', '36', '60', '72000.0']  
]
```

This can be difficult to work with, as the data is not in a convenient format.

There is a better approach...

Reading with csv.DictReader

The same file can be read with the `DictReader` function from the `csv` module.

```
import csv

with open('data/020/windfarm.csv') as csvfile:

    # Attach a DictReader to the file
    windfarm_reader = csv.DictReader(csvfile, skipinitialspace=True)

    # Read each row as a dictionary and store as a list of dictionaries.
    data = [row for row in windfarm_reader]
```

Reading with `csv.DictReader`

The data will now be returned as a list of dictionaries:

```
[
  {
    "id": "WF1355",
    "turbines": "13",
    "height": "53",
    "power": "19500"
  },
  {
    "id": "WF1364",
    "turbines": "3",
    "height": "60",
    "power": "8250"
  }
  // etc
]
```

Using the data

Working with a list of dictionaries is much easier than working with a list of lists.

For example, using a list of lists:

```
# Get the height of the first windfarm  
data[0][2] # not very clear!!
```

vs list of dictionaries:

```
# Get the height of the first windfarm  
data[0]['height']
```

JSON (JavaScript Object Notation)

JSON is a text-based format for storing data that can handle more complex data structures than CSV.

- JSON originated from JavaScript syntax but is **language-independent** and widely used.
- Like Python, JSON uses curly braces to define dictionaries and square brackets to define lists.
- In Data Science, JSON is often used to represent a dataset as a **list of dictionaries**.

JSON example - the windfarm data

```
[  
  {  
    "id": "WF1355",  
    "turbines": 13,  
    "height": 53,  
    "power": 19500  
  },  
  {  
    "id": "WF1364",  
    "turbines": 3,  
    "height": 60,  
    "power": 8250  
  }  
]
```

JSON example - climate data

```
[
  {
    "city": "Amsterdam",
    "country": "Netherlands",
    "monthlyAvg": [
      {
        "high": 7,
        "low": 3,
        "snowDays": 4,
        "rainfall": 68
      },
      {
        "high": 6,
        "low": 3,
        "snowDays": 2,
        "rainfall": 47
      }
      // etc
    ]
  }
  // etc
]
```

Reading a JSON file

The Python Standard Library has a module called `json` that can be used to read and write JSON files. It is super easy to use:

```
import json

# Open the file
with open('data/020/climate.json') as jsonfile:

    # Read the data
    climate_data = json.load(jsonfile)
```

We can now print the first entry in the data:

```
print(climate_data[0])
```

Writing to a JSON file

Writing to a JSON file is equally easy:

```
# Open a file in 'write' mode...  
with open('data/020/climate_copy.json', 'w') as jsonfile:  
  
    # ...use the `dump` method to write the data to the file  
    json.dump(climate_data, jsonfile)
```

JSON supported data types

JSON supports lists and dicts but **it does not support all Python data types**.

For example,

- JSON supports objects (dicts), arrays (lists), strings, numbers, booleans, and null.
- Tuples and sets are not JSON types; convert them to lists before serialising.
- Complex numbers are not supported; convert to strings or to a custom dict (e.g., {"real": ..., "imag": ...}).
- Dictionary keys must be strings in JSON.

These constraints are rarely a problem in practice but can lead to some unexpected behaviour if care is not taken.

The problem with JSON

- Big problem: If storing a dataset as a list of dictionaries, the JSON file requires opening and closing square brackets.
- The file cannot be correctly parsed until the whole file has been read, i.e., the closing square bracket has to be found.
- This is not very convenient if the dataset is very large, or if the data is being streamed from a sensor.
- Solution: The JSON lines format...

JSON lines (`.jsonl`) files

JSON lines files are a variant of JSON files where each line is a separate JSON object.

- Each line is a valid JSON object
- The file as a whole is not a valid JSON object
- e.g., our windfarm data would look like this,

```
{"id": 1355, "turbines": 13, "height": 53, "power": 19500}  
{"id": 1364, "turbines": 3, "height": 60, "power": 8250}
```

Note, each data item is on a separate line, but there are no enclosing square brackets or separating commas.

Reading JSON lines files

The `json` module does not support JSON lines files explicitly, but they can still be easily read and written with a few lines of code.

```
import json

# Open the file
with open('data/020/windfarm.jsonl') as jsonfile:

    # Read each line and store as a list of dictionaries.
    data = [json.loads(line) for line in jsonfile]
```

We are using the `loads` function to convert each line (stored as a string) into the data structure it represents (i.e., a dictionary).

Writing JSON lines files

Writing JSON lines files is also easy,

```
import json

# Open a file in 'write' mode...
with open('data/020/windfarm_copy.jsonl', 'w') as jsonfile:

    # iterate through the data...
    for item in data:

        # ...use the `dump` method to write each item to the file
        json.dump(item, jsonfile)

        # ...and add a newline character so it's on a separate line
        jsonfile.write('\n')
```

Dataset creation good practice

1. Structure and quality

- **Clear structure:** For tabular data, use columns for variables, rows for observations and one table per type of observational unit. [1]
- **Consistent conventions:** Use clear identifiers, types, units, ISO dates and missing-value codes. [2]
- **Quality and reproducibility:** Check errors, duplicates and missing values. Preserve original data and log transformations. [2,3]

Dataset creation good practice

2. Documentation and reuse

- **Purpose and coverage:** Explain intended uses, target population, sampling, exclusions, methods used in collection and labelling. [3,7]
- **Data dictionary:** Define each variable, its type, units, allowed values and missing-value codes. [2]
- **Provenance and limitations:** Record processing history, label uncertainty, known errors, biases and unsuitable uses. [3,4]
- **Support reuse:** Provide a licence, version, persistent identifier, maintainer and update policy. [3,5]

Dataset creation good practice

3. Ethics and stewardship

- **Justified collection:** Document the lawful basis, obtain consent where appropriate and collect only necessary personal data. [3,6]
- **Privacy protection:** Assess identification and linkage risks, including after removing identifiers; set appropriate access controls. [3,5]
- **Fairness and impact:** Assess representation, bias in collection and labelling, label validity, foreseeable harms and affected communities' interests. [4,7]
- **Ongoing stewardship:** Plan retention, corrections and withdrawal; communicate changes to dataset users. [3,6]

References

1. Wickham, H. (2014). *Tidy Data*. *Journal of Statistical Software*.
2. Broman, K. W., & Woo, K. H. (2018). *Data Organization in Spreadsheets*. *The American Statistician*.
3. Gebru, T., et al. (2021). *Datasheets for Datasets*. *Communications of the ACM*.
4. Paullada, A., et al. (2021). *Data and its (dis)contents: A survey of dataset development and use in machine learning research*. *Patterns*.
5. Wilkinson, M. D., et al. (2016). *The FAIR Guiding Principles for scientific data management and stewardship*. *Scientific Data*.
6. Information Commissioner's Office (ICO). *A guide to lawful basis*.
7. Suresh, H., & Gutttag, J. (2021). *A Framework for Understanding Sources of Harm throughout the Machine Learning Life Cycle*. *ACM EAAMO*.

API (Application Programming Interface)

A structured way for an application to request data or functionality from another application.

The most common way to access an API is via HTTP requests. HTTP requests were initially developed to allow web browsers to communicate with web servers, but they can also be used by other programs (like Python scripts) to communicate with remote servers.

HTTP supports different request methods, the most common of which are:

- GET: retrieve data from a server.
- POST: send data to a server.
- PUT: update data on a server.
- DELETE: delete data from a server.

Accessing an API: endpoints and parameters

A web API typically exposes multiple `endpoints`. An endpoint is a specific destination to which an application sends a request to access data or perform an action. For example, sending a `GET` request to `https://pokeapi.co/api/v2/pokemon/pikachu/` retrieves data about Pikachu.

The endpoint will often include `parameters` that can be used to filter or modify the returned data. These parameters are typically added to the endpoint URL as query strings.

The exact endpoint and parameters depend on the API you are using. Always check the API's documentation for the correct URLs and query options.

When making an `HTTP GET` request, the server receives the request, processes it, and returns a `response` that includes a `status code` (success or error) and, if successful, the requested `data`.

API requests: response status codes

- 200: OK (the request was successful).
- 400: Bad Request (the request was malformed).
- 401: Unauthorized (you need to authenticate to access the requested resource).
- 403: Forbidden (you do not have permission to access the requested resource).
- 404: Not Found (the requested resource could not be found).
- 500: Internal Server Error (the server encountered an error).

API requests: response data format

- **JSON**: most common format; lightweight, easy to read and write.
- **XML (eXtensible Markup Language)**: a markup language that is used to store and transport data.
- **CSV**: simple, common for tabular data.

What next?

- Finish today's tutorials at your own pace:
 - `/materials/tutorials/020_Reading_Data.ipynb`
 - `/materials/tutorials/025_Obtaining_Data_via_an_API.ipynb`
- Come to the lab tomorrow; we will read and process a dataset
 - `/materials/labs/020_reading_data.ipynb`

