CS410: Principles and Techniques of Data Science

Module 5: Wrangling Files
https://drive.google.com/drive/folders/1aV6Er2WUBoav3Y-gAt3we5uDy23e_gJg?usp=sharing

Introduction

Before you can work with data, you need to know:

- How much data do you have?
- How is the source file formatted?

Answers to these questions can be very helpful.

If your file is too large, you may need special approaches to read it in.

If your file isn't formatted the way you expect, you may run into bad values after loading it into a dataframe.

Two datasets as examples:

- A government survey about drug abuse and
- Administrative data from the City of San Francisco about restaurant inspections.

Drug Abuse Warning Network (DAWN)

- DAWN - national healthcare survey that monitors trends in drug abuse
- Administered by the U.S. Substance Abuse and Medical Health Services Administration (SAMHSA).

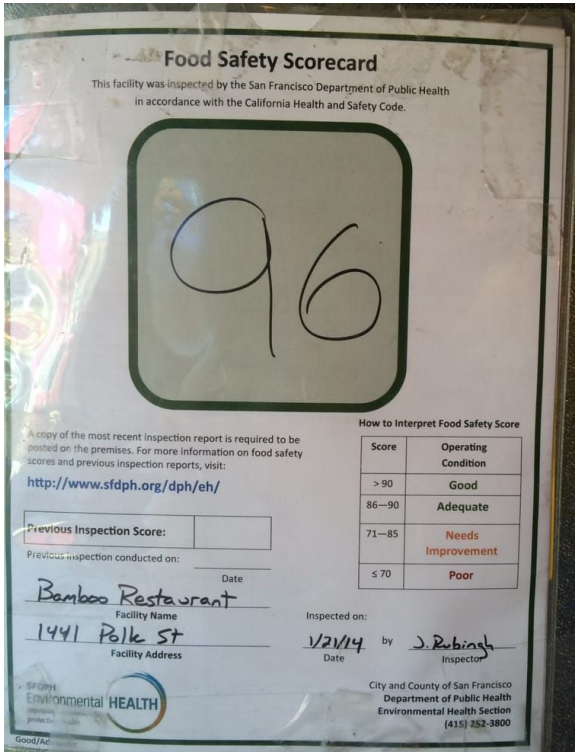
Target population: all drug-related, emergency-room visits in the U.S.

San Francisco Restaurant Food Safety

The San Francisco Department of Public Health routinely makes unannounced visits to restaurants and inspects them for food safety. The inspector calculates a score based on the violations observed and provides descriptions of the violations that were found.

Target population: All restaurants in the City of San Francisco.

Data: Restaurant inspections that were conducted between 2013 and 2016. Some restaurants have multiple inspections in a year, and not all of the 7000+ restaurants are inspected annually.



A food safety scorecard displayed in restaurant. Scores range between 0 and 100.

File Formats

File format describes how data are stored on the computer.

Understanding the file format helps us figure out how to read the data into Python.

- Delimited Format
- Fixed-width Format
- Hierarchical Format
- Loosely Structured Formats

Delimited Format

- Delimited formats use a specific character to separate data values.
- Separators are either: a comma (Comma-Separated-Values or CSV for short), a tab (Tab-Separated Values or TSV), white-space, or a colon.
- The first line of these files often contains the names of the table's columns/features.

```
"business_id","score","date","type"
19,"94","20160513","routine"
19,"94","20171211","routine"
24,"98","20171101","routine"
24,"98","20161005","routine"
```

```
Sepal length    Sepal width     Petal length    Petal width     Species
5.1     3.5     1.4     0.2     I. setosa
4.9     3.0     1.4     0.2     I. setosa
4.7     3.2     1.3     0.2     I. setosa
4.6     3.1     1.5     0.2     I. setosa
5.0     3.6     1.4     0.2     I. setosa
```

Delimited Format

The San Francisco restaurant scores are stored in CSV-formatted files.

In Python, the built-in `pathlib` library has a useful `Path` object to specify paths to files and folders that work across platforms. The data are stored in the file `data/inspections.csv`.

The `Path` object has many useful methods, such as `open()`:

```
from pathlib import Path

# Create a Path pointing to our data file
insp_path = Path() / 'data' / 'inspections.csv'

with insp_path.open() as f:
    # Display first five lines of file
    for _ in range(5):
        print(f.readline(), end='')
```

```
"business_id","score","date","type"
19,"94","20160513","routine"
19,"94","20171211","routine"
24,"98","20171101","routine"
24,"98","20161005","routine"
```

Delimited Format

Paths are tricky when working across different operating systems (OS).

Eg: a typical path in Windows might look like `C:\files\data.csv`
a path in Unix or Mac OS might look like `~/files/data.csv`.

Because of this, code that works on one OS can fail to run on other operating systems.

The `pathlib` Python library was created to avoid OS-specific path issues.

Fixed-width Format

The fixed-width format (FWF) does not use delimiters to separate data values.

The values for a specific field appear in the exact same position in each line.

```
dawn_path = Path() / 'data' / 'DAWN-Data.txt'
width = 65
with dawn_path.open() as f:
    for _ in range(5):
        print(f.readline()[:width])
```

```
1 2251082    .9426354082   3 4 1 2201141 2 865 105 1102005 1
2 2291292   5.9920106887   911 1 3201134 12077  81  82 283-8
3 7 7 251   4.7231718669   611 2 2201143 12313   1  12  -7-8
410 8 292   4.0801470012   6 2 1 3201122 1 234 358  99 215 2
5 122 942   5.1777093467  10 6 1 3201134 3 865 105 1102005 1
```

SAMHSA provides a 2,000-page codebook with all of the information needed to read the file.

Example: “age” field appears in positions 34-35.

Filename Extensions

- A widely adopted convention is to use the filename extension, such as `.csv`, `.tsv`, and `.txt`, to indicate the format of the contents of the file.
- File names that end with `.csv` are expected to contain comma-separated values, `.tsv` tab-separated values, and `.txt` generally is plain text without a particular format.
- These extension names are only suggestions. Even if a file has a `.csv` extension, the actual contents might not be formatted properly! It's good practice to inspect the contents of the file before loading it into a data frame.
- If the file is not too large, you can open and examine it with a plain text editor. Otherwise, you view a couple of lines using `.readline()`.

Hierarchical Formats

- Hierarchical data formats store data with a nested structure.
- Eg: the JavaScript Object Format (JSON) is a common format used for communication by web servers. JSON files have a hierarchical structure with keys and values similar to a Python dictionary.

```
{"widget": {
    "debug": "on",
    "window": {
        "title": "Sample Konfabulator Widget",
        "name": "main_window",
        "width": 500,
        "height": 500
    },
    "image": {
        "src": "Images/Sun.png",
        "name": "sun1",
        "hOffset": 250,
        "vOffset": 250,
        "alignment": "center"
    },
    "text": {
        "data": "Click Here",
        "size": 36,
        "style": "bold",
        "name": "text1",
        "hOffset": 250,
        "vOffset": 100,
        "alignment": "center",
        "onMouseUp": "sun1.opacity = (sun1.opacity / 100) * 90;"
    }
}}
```

Loosely Structured Formats

Web logs, instrument readings, and program logs typically provide data in plain text. It contains information such as the date and time and type of request made to the Web site.

```
169.237.46.168 - -
[26/Jan/2004:10:47:58 -0800]"GET /stat141/Winter04 HTTP/1.1" 301 328
"http://anson.ucdavis.edu/courses"
"Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.1.4322)"
```

Single recording of a measurement taken with a wireless device. The device reports the timestamp, identifier, location of the device, and the signal strengths that it picks up from other devices.

```
t=1139644637174;id=00:02:2D:21:0F:33;pos=2.0,0.0,0.0;degree=45.5;
00:14:bf:b1:97:8a=-33,2437000000,3;00:14:bf:b1:97:8a=-38,2437000000,3;
```

File Encoding

- Modern computers store data as long sequences of *bits*: `0`s and `1`s.
- ASCII: tell the computer how to translate between bits and actual text.

  Eg: In ASCII, the bits `100 001` stand for the letter `A`, and `100 010` for `B`.

  ASCII = uppercase + lowercase English letters + numbers + punctuation symbols + spaces

  ASCII encoding is not sufficient to represent a lot of special characters and characters from other languages.
- Common encodings for documents and Web pages are Latin-1 (ISO-8859-1) and UTF-8.

  UTF-8 has over one million characters, and is backwards compatible with ASCII, meaning that it uses the same representation for English letters, numbers, and punctuation as ASCII.

File Encoding

- When we have a text file, we usually need to figure out its encoding. If we choose the wrong encoding to read in a file, Python either reads incorrect values or errors.
- The best way to find the encoding is by checking the data's documentation which often explicitly says what the encoding is.
- When we don't know what the encoding is, we have to make a guess. The `chardet` package has a function called `detect()` that infers a file's encoding. Since these guesses are imperfect, the function also returns a confidence between 0 and 1.

File Encoding

```
import chardet
line = '{:<25} {:<10} {}'.format
# for each file, print its name, encoding & confidence in the encoding
print(line('File', 'Encoding', 'Confidence'))
for filepath in Path('data').glob('*'):
    result = chardet.detect(filepath.read_bytes())
     print(line(str(filepath), result['encoding'], result['confidence']))
```

```
File                      Encoding   Confidence
data/inspections.csv      ascii      1.0
data/co2_mm_mlo.txt       ascii      1.0
data/violations.csv       ascii      1.0
data/DAWN-Data.txt        ascii      1.0
data/legend.csv           ascii      1.0
data/businesses.csv       ISO-8859-1 0.73
```

File Encoding

The detection function is quite certain that all but one of the files are ASCII encoded. The exception is `businesses.csv`, which appears to have an ISO-8859-1 encoding. We run into trouble, if we ignore this encoding and try to read the business file into Pandas without specifying the special encoding.

```
# naively reads file without considering encoding
>>> pd.read_csv('data/businesses.csv')
[...stack trace omitted...]
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xd1 in
 position 8: invalid continuation byte
```

To successfully read the data, we must specify the ISO-8859-1 encoding.

```
bus = pd.read_csv('data/businesses.csv',  encoding='ISO-8859-1')
```

	business_id	name	address	postal_code
0	19	NRGIZE LIFESTYLE CAFE	1200 VAN NESS AVE, 3RD FLOOR	94109
1	24	OMNI S.F. HOTEL - 2ND FLOOR PANTRY	500 CALIFORNIA ST, 2ND FLOOR	94104
2	31	NORMAN'S ICE CREAM AND FREEZES	2801 LEAVENWORTH ST	94133
3	45	CHARLIE'S DELI CAFE	3202 FOLSOM ST	94110

File Size

- Computers have finite resources. We want to make sure not to exceed the computer's limits while working with data, and we might examine a file differently, depending on its size.

Dataset is small => text editor or a spreadsheet can be convenient to look at the data.

Huge dataset => a more programmatic exploration or even distributed computing tools may be needed.

File Size

- In many situations, we analyze datasets downloaded from the Internet. These files reside on the computer's *disk storage*.
- To use Python to explore and manipulate the data, we need to read the data into the computer's *memory*, also known as random access memory (RAM).
- A computer's RAM is typically much smaller than a computer's disk storage. Eg: one computer model released in 2018 had 32 times more disk storage than RAM.
- Unfortunately, this means that data files can often be much bigger than what is feasible to read into memory.

File Size

To make sure the files we are of manageable size, we can use the built-in `os` library.

```
from pathlib import Path
import numpy as np
import os
kib = 1024
line = '{:<25} {}'.format
print(line('File', 'Size (KiB)'))
for filepath in Path('data').glob('*'):
    size = os.path.getsize(filepath)
     print(line(str(filepath), np.round(size / kib)))
```

```
File                      Size (KiB)
data/inspections.csv      455.0
data/co2_mm_mlo.txt       50.0
data/violations.csv       3639.0
data/DAWN-Data.txt        273531.0
data/legend.csv           0.0
data/businesses.csv       645.0
```

File Size

- The `businesses.csv` file takes up 645 KiB on disk, making it well within the memory capacities of most systems.
- Although the `violations.csv` file takes up 3.6 MiB of disk storage, most machines can easily read it into a Pandas DataFrame too.
- But `DAWN-Data.txt`, which contains the DAWN survey data, is much larger.
  - Takes up nearly 270 MiB of disk storage, and while some computers can work with this file in memory, it can slow down other systems.
  - To make this data more manageable in Python, we can load in a subset of the columns rather than all of them.

```
File                    Size (KiB)
data/inspections.csv    455.0
data/co2_mm_mlo.txt     50.0
data/violations.csv     3639.0
data/DAWN-Data.txt      273531.0
data/legend.csv         0.0
data/businesses.csv     645.0
```

File Size

Sometimes we are interested in the total size of a folder instead of the size of individual files.

Eg: if we have one file of inspections for each month in a year, we might like to see whether we can combine all the data into a single data frame.

```
mib = 1024**2

total = 0

for filepath in Path('data').glob('*'):

    total += os.path.getsize(filepath) / mib

print(f'The data/ folder contains {total:.2f} MiB')
```

```
The data/ folder contains 271.80 MiB
```

File Size

- Reading in a file using `pandas` usually requires at least five times the available memory as the file size. For example, reading in a 1 GiB file will typically require at least 5 GiB of available memory.
- Memory is shared by all programs running on a computer, including the operating system, web browsers, and Jupyter notebook itself. A computer with 4 GiB total RAM might have only 1 GiB available RAM with many applications running. With 1 GiB available RAM, it is unlikely that `pandas` will be able to read in a 1 GiB file.

Working with Large Datasets

Some data are large enough that even top-of-the-line computers can't read the data directly into memory. This is a common scenario in scientific domains like astronomy, where telescopes capture many large images of space. It's also common for companies that have lots of users.

Subset The Data

Rather than loading in the entire source file, we can either select a specific part of it (e.g. one day's worth of data), or we can randomly sample the dataset. Downside is that this approach loses many of the benefits of analyzing a large dataset.

Use a Database System

Relational database management systems (RDBMS) were specifically designed to store large datasets and manipulate them using SQL queries.

Table Shape and Granularity

granularity: what each row in the table represents

```
bus = pd.read_csv('data/businesses.csv', encoding='ISO-8859-1')
insp = pd.read_csv("data/inspections.csv")
viol = pd.read_csv("data/violations.csv")

print(" Businesses shape:", bus.shape)
print("Inspections shape:", insp.shape)
print(" Violations shape:", viol.shape)
```

```
 Businesses shape: (6406, 9)
Inspections shape: (14222, 4)
 Violations shape: (39042, 3)
```

Table Shape and Granularity

```
Businesses shape: (6406, 9)
```

The businesses table has 6406 rows and 9 columns.

Now, let's find the granularity of this table by understanding what each row represents.

	business_id	name	address	city	...	postal_code	latitude	longitude	phone_
0	19	NRGIZE LIFESTYLE CAFE	1200 VAN NESS AVE, 3RD FLOOR	San Francisco	...	94109	37.79	-122.42	+14157
1	24	OMNI S.F. HOTEL - 2ND FLOOR PANTRY	500 CALIFORNIA ST, 2ND FLOOR	San Francisco	...	94104	37.79	-122.40	+14156

2 rows × 9 columns

Does each record represents a single restaurant?

Table Shape and Granularity

We can’t tell from just two records whether or not this is the case.

The field name `business_id` implies that it is the unique identifier for the business (restaurant).

We can confirm this by checking whether the number of records in `bus` matches the number of unique values in `business_id`.

```
print("Number of records:", len(bus))
print("Number of unique business ids:", len(bus['business_id'].unique()))
```

```
Number of records: 6406
Number of unique business ids: 6406
```

The number of unique `business_id`s matches the number of rows in the table.

Summary

- Data wrangling is an essential part of data analysis. Without it, we risk overlooking problems in data that can have major consequences for future analysis.
- We covered an important first step in data wrangling: reading data from a plain text source file into a Python DataFrame.
- We introduced different types of file formats and encodings, and we wrote code that can read data from these formats. We checked the size of source files and considered alternative tools for working with large datasets.

Understanding the shape and granularity of a table gives us insight into what a row in a data table represents.

Knowing your table's granularity is a first step to cleaning your data, and it informs you of how to analyze your data.

THANK YOU!