6 Big Data File Formats Compared, Pt. 2

Made with Obsidian

Type blog | Category big-data | Technologies Python | Website Post Link

In the first part of this 3-article series, we introduced the concepts of **columnar file formats** & **row-based file formats**. We also defined **serialization** and **deserialization** and provided an overview of six relevant Big Data file formats. Finally, we went over some examples involving writing different objects to these file formats using Python.

In this section, we will focus on **reading** the files we created.

We'll be using Python scripts which can be found in the Blog Article Repo.

§

Table of Contents

§

Preparing the files

On our last session we generated a total of 12 files:

File Name	Format	Method Used
`01_dataset_method_1.csv`	CSV	`numpy.tofile()`
`02_dataset_method_2.csv`	CSV	`numpy.savetext()`
`03_dataset_method_3.csv`	CSV	`pandas.DataFrame.to_csv()`
`04_dataset_method_1.txt`	TXT	`numpy.savetext()`
`05_dataset_method_2.txt`	TXT	`pandas.DataFrame.to_csv()`
`06_dataset_method_1.feather`	Feather	`pandas.DataFrame.to_feather()`
`07_dataset_method_2.parquet`	Parquet	`pandas.DataFrame.to_parquet()`
`08_dataset_method_2.parquet`	Parquet	`pandas.DataFrame.to_parquet()`
`09_dataset_method_3.parquet`	Parquet	`pandas.DataFrame.to_parquet()`
`10_dataset_method_1.avro`	Avro	`fastavro`
`11_dataset_method_1.pickle`	Pickle	`pickle.dump()`
`12_dataset_method_2.pickle`	Pickle	`pickle.dumps()`

Table 1. Files Written On Part 1 Of Article Series

We will only be reading outputs 02, 05, 06, 07, 08, 09, 10 & 11.

Files can be found in the blog article repo `outputs` folder.

§

Reading with Python

As a first step, we will import all the required modules:

Code

```
import csv
import numpy as np
import pandas as pd
from fastavro import reader
import pickle
```

If we don't have the `pyarrow` library already installed, we can do so since we'll be needing it for `.feather` & `.parquet` file format reading.

Code

```
pip install pyarrow
```

1. CSV

There are two primary methods for reading a CSV file using Python:

1.1 Using `csv.reader()`

This method uses the Python file handler, and creates a `reader` object by using the `csv.reader()` method. We can then use the `csv.reader()` method as a row iterable in order to build a `numpy.ndarray()` object, from a list of lists object `lol`:

Code

```
# Declare an empty list object
lol = []

# Define the file handler, and open in read mode
with open("outputs/02_dataset_method_2.csv", 'r') as file:
  csvreader = csv.reader(file)
  # Iterate over rows and append to list
  for row in csvreader:
      lol.append(row)

# Close the Text Wrapper object
file.close()

# Check the resulting object's type
type(lol)
```

Output

```
list
```

We can then build a `numpy.ndarray` object `arr` from our list of lists object:

Code

```
# Build a numpy.ndarray object from a list of lists
arr = np.array(lol)

# Check the resulting object's type
type(arr)

# Print the object
arr
```

Output

```
numpy.ndarray

array([['Name', 'Age', 'Occupation', 'Country', 'State', 'City'],
       ['Joe', '20', 'Student', 'United States', 'Kansas', 'Kansas City'],
       ['Chloe', '37', 'Detective', 'United States', 'California',
        'Los Angeles'],
       ['Dan', '39', 'Detective', 'United States', 'California',
        'Los Angeles']], dtype='<U13')
```

If we wanted to, we could also convert our `numpy.ndarray` object `arr`, to a `pandas.DataFrame` object `df`, by first specifying row 1 through the end as data, & row 0 as the header:

Code

```
# Build a pandas.DataFrame object from numpy.ndarray object
df = pd.DataFrame(arr[1:], columns = arr[0])

# Check the resulting object's type
type(df)

# Print the object
df
```

Output

```
pandas.core.frame.DataFrame

    Name Age Occupation        Country       State         City
0    Joe  20    Student  United States      Kansas  Kansas City
1  Chloe  37  Detective  United States  California  Los Angeles
2    Dan  39  Detective  United States  California  Los Angeles
```

As we can see, this method involves a fair amount of steps that could be easily avoided by using the `pandas.read_csv()` method:

1.2 Using `pandas.read_csv()`

As with writing, this method is by far the most used if we aim to get a `pandas.DataFrame()` object in return.

With one line of code, we can directly read the `.csv` file into a `pandas.DataFrame()` object:

Code

```
# Read the csv and import to pandas.DataFrame object
df = pd.read_csv("outputs/02_dataset_method_2.csv")

# Check the resulting object's type
type(df)

# Print the object
df
```

As we can see, we get the exact same result as with the previous method:

Output

```
pandas.core.frame.DataFrame

    Name Age Occupation        Country       State         City
0    Joe  20    Student  United States      Kansas  Kansas City
1  Chloe  37  Detective  United States  California  Los Angeles
2    Dan  39  Detective  United States  California  Los Angeles
```

And the best thing is, we don't even have to specify the first row as our header, since the default behavior for this method handles that for us.

2. TXT

There are two primary methods for reading a TXT file using Python:

2.1 Using the built-in Python file handler

Similar to the `csv.reader()` methodology previously shown, we can make use of the built-in Python file handler:

Code

```python
# Declare an empty list object
lol = []

# Define the file handler, and open in read mode
with open("outputs/04_dataset_method_1.txt", 'r') as file:
    # Read the entire content and split with newlines
    content = file.read().split('\n')

    # For every newline, split in tab delimitors
    for entry in content:
        entry = entry.split('\t')
        # Append each splitted entry in list of lists
        lol.append(entry)

# Close the Text Wrapper object
file.close()

# Return the results
lol
```

If we recall from the first part of this article series, we mentioned that it was important to remember which delimiter we were using, especially when working with TXT files. This is because when we open a `.txt` file, at least with this method and the next one we will review, we need to know the delimiter used to write it in order to parse the content properly.

In our case, a newline `\n` delimiter was used for specifying rows, and a tab `\t` delimiter was used for specifying entries. This is why we needed to include the additional `.split()` methods for each case.

Output

```
['Name', 'Age', 'Occupation', 'Country', 'State', 'City']
['Joe', '20', 'Student', 'United States', 'Kansas', 'Kansas City']
['Chloe', '37', 'Detective', 'United States', 'California', 'Los Angeles']
['Dan', '39', 'Detective', 'United States', 'California', 'Los Angeles']
['']
```

If we look closely at the output, we can see an empty list at the end of our list of lists object `lol`. This is because the split method splits the object into two parts, and in our case, the last line was split, with the first part being the actual list and the second being an empty list since we had no additional data.

We can take care of this by simply removing the last entry of our `lol` object:

Code

```python
# Remove last entry using inplace method .pop()
lol.pop()
```

We can now simply convert the list of lists object `lol` to a `numpy.ndarray` object:

Code

```
# Convert list of lists to
arr = np.array(lol)

# Print our output
arr
```

Output

```
Name    Age Occupation  Country State   City
Joe 20  Student United States   Kansas  Kansas City
Chloe   37  Detective   United States   California  Los Angeles
Dan 39  Detective   United States   California  Los Angeles
```

If we wanted to, we could also convert our `numpy.ndarray` object `arr`, to a `pandas.DataFrame` object `df`, by first specifying rows 1 through the end as data, & row 0 as header:

Code

```
# Build a pandas.DataFrame object from numpy.ndarray object
df = pd.DataFrame(arr[1:], columns = arr[0])

# Check the resulting object's type
type(df)

# Print the object
df
```

Output

```
pandas.core.frame.DataFrame

    Name Age Occupation        Country       State         City
0    Joe  20    Student  United States      Kansas  Kansas City
1  Chloe  37  Detective  United States  California  Los Angeles
2    Dan  39  Detective  United States  California  Los Angeles
```

As we can see, this method involves a fair amount of steps that could be easily avoided by using the `pandas.read_csv()` method:

2.2 Using `pandas.read_csv()`

Similar to reading a `.csv` file, we can use the `pandas.read_csv()` method to read a `.txt` file:

Code

```
# Read the txt file and import to pandas.DataFrame object
df = pd.read_csv("outputs/04_dataset_method_1.txt", sep = '\t')

# Check the resulting object's type
type(df)

# Print the object
df
```

If we pay close attention to the code, a `sep` parameter is included. In our case, we know that the `.txt` file was written using a tab `\t` separator to denote each entry, so we specify the entry separator as a tab.

Output

```
pandas.core.frame.DataFrame

    Name Age Occupation        Country       State        City
0    Joe  20    Student  United States      Kansas  Kansas City
1  Chloe  37  Detective  United States  California  Los Angeles
2    Dan  39  Detective  United States  California  Los Angeles
```

3. Feather

There is one method for reading a Feather file using Python:

3.1 Using `pandas.read_feather()`

Analogous to the `df.to_feather()` method, we have a way to read `.feather` files using Pandas:

Code

```
# Read the feather file and import to pandas.DataFrame object
df = pd.read_feather("outputs/06_dataset_method_1.feather")
```

4. Parquet

There is one method for reading a Parquet file using Python:

4.1 Using `pandas.read_parquet()` for non-partitioned files

Analogous to the `pandas.Dataframe.to_parquet()` method, we have a way to read non-partitioned `.parquet` files using Pandas:

Code

```
# Read the parquet non-partitioned file and import to pandas.DataFrame object
df = pd.read_parquet("outputs/06_dataset_method_1.feather")
```

4.2 Using `pandas.read_parquet()` for single and multi-partitioned files

We can also read single & multi-partitioned files using the same method without the need to specify any additional parameters:

Code

```
# Read the parquet single-partitioned file and import to pandas.DataFrame object
df = pd.read_parquet("outputs/07_dataset_method_2.feather")

# Read the parquet multi-partitioned file and import to pandas.DataFrame object
df = pd.read_parquet("outputs/08_dataset_method_3.feather")
```

This makes reading `.parquet` files seamless with Pandas, whichever the partition schema is.

5. Avro

There is one method for reading a Avro file using Python:

5.1 Using `fastavro` `reader`

If we recall from the first part of this article series, we mentioned that in order to write a `.avro` file, we first needed to convert our data set into dictionaries consisting of key-value pairs (*one dictionary per row*). Then, we needed to save our dictionaries as a list.

The same applies when attempting to read a `.avro` file:

```
# Declare an empty list of dictionaries
lod = []

# Use the Python file handler along with the fastavro reader method
with open('outputs/10_dataset_method_1.avro', 'rb') as fo:
    avro_reader = reader(fo)
    for record in avro_reader:
        lod.append(record)

# Close the BufferedReader object
fo.close()

# Convert list of dictionaries to DataFrame
df = pd.DataFrame.from_dict(lod)
```

If we pay close attention to the code above, we specify a reading mode parameter `rb`, meaning read in binary mode.

Then, we iterate over the `fastavro` `_read.reader()` object, and append each dictionary (*row*) to our list of dictionaries `lod`.

Finally, we convert our list of dictionaries to a `pandas.core.frame.DataFrame` object.

6. Pickle

There are two methods for reading a Pickle file using Python, but we will only be covering one:

6.1 Using `pickle.load()` to read from an open file

Code

```python
# Use the Python file handler
with open('outputs/11_dataset_method_1.pickle', 'rb') as file:
    my_pickled_object = pickle.load(file)

# Close the BufferedReader object
file.close()

# Print the deserialized object
print(my_pickled_object)
```

The great thing about serializing and deserializing using `.pickle` file formats is that when reading a `.pickle` file, we will get the exact same object we wrote.

Output

```
{'Name': 'Joe', 'Age': 20, 'Occupation': 'Student', 'Country': 'United States', 'State':
'Kansas', 'City': 'Kansas City'}
{'Name': 'Chloe', 'Age': 37, 'Occupation': 'Detective', 'Country': 'United States', 'State':
'California', 'City': 'Los Angeles'}
{'Name': 'Dan', 'Age': 39, 'Occupation': 'Detective', 'Country': 'United States', 'State':
'California', 'City': 'Los Angeles'}
```

§

Conclusions

We've reviewed different reading methods for six file formats. We've also specified which method is best for a given application, whether we're working with `numpy.ndarray` objects, `pandas.DataFrame` objects, or other Python objects such as tuples, lists & dictionaries.

Now that we know how to write & read these formats using Python, it's time to move on to comparing them.

§

References

- Geeks for Geeks, Working with csv files in Python
- Python Tutorial, Python Read Text Files
- Earthly, How To Read A CSV File In Python
- Build Media, Fastavro Documentation

§

Copyright