Co-funded by the
Erasmus+ Programme
of the European Union

DeCAIR

Data Loading, Storage and File Formats

Prof. Gheith Abandah

Reference

- **Chapter 6**
- Wes McKinney, **Python for Data Analysis**: Data Wrangling with Pandas, NumPy, and IPython, O'Reilly Media, 2nd Edition, 2018.
  - Material: https://github.com/wesm/pydata-book

Outline

6.1 Reading and Writing Data in Text Format

6.2 Binary Data Formats

6.3 Interacting with Web APIs

6.4 Interacting with Databases

Outline

6.1 Reading and Writing Data in Text Format

6.2 Binary Data Formats

6.3 Interacting with Web APIs

6.4 Interacting with Databases

- Parsing Functions
- Reading Text Files in Pieces
- Writing Data to Text Format
- JSON Data
- XML and HTML: Web Scraping

Parsing Functions in pandas

Function	Description
`read_csv`	Load delimited data from a file, URL, or file-like object; use comma as default delimiter
`read_table`	Load delimited data from a file, URL, or file-like object; use tab (`'\t'`) as default delimiter
`read_fwf`	Read data in fixed-width column format (i.e., no delimiters)
`read_clipboard`	Version of `read_table` that reads data from the clipboard; useful for converting tables from web pages
`read_excel`	Read tabular data from an Excel XLS or XLSX file
`read_hdf`	Read HDF5 files written by pandas
`read_html`	Read all tables found in the given HTML document
`read_json`	Read data from a JSON (JavaScript Object Notation) string representation
`read_msgpack`	Read pandas data encoded using the MessagePack binary format
`read_pickle`	Read an arbitrary object stored in Python pickle format

Most used

Parsing Functions in pandas – cont.

Function	Description
`read_sas`	Read a SAS dataset stored in one of the SAS system's custom storage formats
`read_sql`	Read the results of a SQL query (using SQLAlchemy) as a pandas DataFrame
`read_stata`	Read a dataset from Stata file format
`read_feather`	Read the Feather binary file format

6.1 Reading and Writing Data in Text Format

- These parsing functions take **optional arguments** that fall into the following categories:
  - **Indexing**
  - **Type inference and data conversion**
  - **Datetime parsing**
  - **Iterating**
  - **Unclean data issues**
- **Many options**, so refer to the **online documentation** for **complex cases**.

Comma-Separated (CSV) Text Files

```
ex1.csv
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

```
   0  1  2  3      4
0  1  2  3  4 hello
…
```

- For files with **no headers**, ask pandas to **assign default** column names, or **specify names**.

```
pd.read_csv('ex2.csv',
            header=None)

pd.read_csv('ex2.csv',
        names=['a', 'b', 'c', 'd',
        'message'])
```

```
df = pd.read_csv('ex1.csv')
df
```

```
   a  b  c  d message
0  1  2  3  4   hello
1  5  6  7  8   world
2  9 10 11 12     foo
```

Alternative:

```
pd.read_table('ex1.csv', sep=',')
```

Comma-Separated (CSV) Text Files

- You can use **one of the file columns** as **index**.
- How to handle fields separated by a **variable amount of whitespace**?

ex3.csv

```
            A         B         C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
```

Must be index, why?

```
names = ['a', 'b', 'c', 'd',
         'message']

pd.read_csv('ex2.csv', names=names,
            index_col='message')
```

```
         a   b   c   d
message
hello    1   2   3   4
world    5   6   7   8
foo      9  10  11  12
```

Regular expression for one or more white space

```
pd.read_table('ex3.txt', sep='\s+')
```

Comma-Separated (CSV) Text Files

- **Missing data** is usually either **not present** (empty string) or marked by some **sentinel value**.
- Can Specify the sentinel values.

ex5.csv

```
something,a,b,c,d,msg
one,1,2,3,4,NA
two,5,6,,8,foo
```

```
pd.read_csv('ex5.csv')
```

```
  something  a  b    c  d   msg
0       one  1  2  3.0  4   NaN
1       two  5  6  NaN  8   foo
```

```
sentinels = {'msg': ['foo', 'NA'],
             'something': ['two']}

pd.read_csv('examples/ex5.csv',
            na_values=sentinels)
```

```
  something  a  b    c  d   msg
0       one  1  2  3.0  4   NaN
1       NaN  5  6  NaN  8   NaN
```

Reading Text Files in Pieces

- If you want to **read a small number of rows**, use **`nrows`**.
- To **read a file in pieces**, specify a **`chunksize`** of rows.
- **Iterate** on the returned parser object **to aggregate** the value counts in the **`'key'`** column.
- There is also **`chunker.get_chunk(n)`**.

```
pd.read_csv('ex6.csv', nrows=5)

chunker = pd.read_csv('ex6.csv',
                      chunksize=1000)
chunker
<pandas.io.parsers.TextFileReader
at 0x7f6b1e2672e8>

tot = pd.Series([])
for c in chunker:
    tot = tot.add(
            c['key'].value_counts(),
            fill_value=0)

tot = tot.sort_values(
                      ascending=False)
```

Writing Data to Text Format

- We can **write** the data out **to a comma-separated file**.
- Useful options: **sep**, **na_rep**, **index**, and **header**.
- You can also write only a **subset** of the columns, and in an **order** of your choosing.

```
data.to_csv('out.csv')

data.to_csv(sys.stdout,
            sep='|' ,
            na_rep='NULL')
|something|a|b|c|d|message
0|one|1|2|3.0|4|NULL

data.to_csv(sys.stdout,
            index=False,
            header=False)

data.to_csv(sys.stdout,
          columns=['a', 'b', 'c'])
```

JSON Data

- **JSON** (short for **JavaScript Object Notation**) is a standard formats for sending data.
- It is a **free-form** data format. Example:

```
{"name": "Wes",
"places_lived": ["United States", "Spain"],
"pet": null,
"siblings": [{"name": "Scott", "age": 30},
             {"name": "Katie", "age": 38}]
}
```

Call it `j_str`

Nearly valid Python code. **Exceptions**: The null value is `null`. **Disallowing trailing commas** at the end of lists. All of the **keys** in an object must be **strings**.

JSON Data

- **Python has built in JSON support**.
- To convert a **JSON string to Python form**, use `json.loads`.
- `json.dumps` converts a **Python object to JSON**.
- Python `dict` to `DataFrame`.

Also `json.load(fp)`

```
result = json.loads(j_str)

asjson = json.dumps(result)
```

Also `json.dump(fp)`

```
siblings = pd.DataFrame(
           result['siblings'],
           columns=['name', 'age'])
siblings
    name age
0  Scott  30
1  Katie  38
```

JSON Data

- The default options for **pandas.read_json** assume that each object in the JSON array is a row in the table.
- To export data from pandas to JSON, use the **to_json**.

example.json

```
[{"a": 1, "b": 2, "c": 3},
 {"a": 4, "b": 5, "c": 6},
 {"a": 7, "b": 8, "c": 9}]
```

```
data = pd.read_json('example.json')
data
```

```
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

```
print(data.to_json(
          orient='records'))
```

```
[{"a":1,"b":2,"c":3},{"a":4,"b":5,"
c":6}]
```

XML and HTML: Web Scraping

- Given the **html** document from the US FDIC list for bank failures, find the **years with most bank failures**.
- Pandas has `read_html` that returns a list of DataFrames.
- We need to extract years from the 'Closing Date' column.

```
conda install lxml
pip install beautifulsoup4 html5lib

tables = pd.read_html(
       'fdic_failed_bank_list.html')
failures = tables[0]

close_timestamps = pd.to_datetime(
           failures['Closing Date'])
close_timestamps.dt.year.
                        value_counts()

2010 157
2009 140
2011 92
…
```

Outline

6.1 Reading and Writing Data in Text Format

6.2 Binary Data Formats

- Pickle
- Using HDF5 Format
- Reading Microsoft Excel Files

6.3 Interacting with Web APIs

6.4 Interacting with Databases

6.2 Binary Data Formats

- Python's has built-in **pickle** serialization.
- **Serialization** is converting an object in memory to a byte stream that can be stored on disk or sent over a network.
- Python's pickle package has **dump** and **load**.
- Good for short-term storage.

```
import pickle

dogs_dict = { 'Ozzy': 3, 'Filou': 8,
 'Luna': 5, 'Skippy': 10, 'Barco': 12,
 'Balou': 9, 'Laika': 16 }

filename = 'dogs'
outfile = open(filename, 'wb')
pickle.dump(dogs_dict, outfile)
outfile.close()

infile = open(filename, 'rb')
new_dict = pickle.load(infile)
infile.close()
```

6.2 Binary Data Formats

- All pandas objects have **`to_pickle`**.

- The reverse is **`read_pickle`**.

```
frame = pd.DataFrame(
        np.arange(9).reshape((3, 3)))
frame.to_pickle('frame_pickle')
```

```
pd.read_pickle('frame_pickle')
```

```
   0  1  2
0  0  1  2
1  3  4  5
2  6  7  8
```

Using HDF5 Format

- The **hierarchical data format** is efficient and cross platform.
- Pandas has built-in support for HDF5.
- Use **to_hdf** and **read_hdf** to access one or more pandas objects in an HDF5 file.

```
df = pd.DataFrame({'A': [1, 2, 3],
                   'B': [4, 5, 6]})
df.to_hdf('data.h5', key='df1',
          mode='w')

s = pd.Series([1, 2, 3, 4])
s.to_hdf('data.h5', key='s1')

pd.read_hdf('data.h5', 'df1')
```

```
   A  B
0  1  4
1  2  5
2  3  6
```

Using HDF5 Format

- The **HDFStore** class works like a **dict** and handles the low-level details for writing and retrieving.

```
frame = pd.DataFrame({'a':
                      np.random.randn(100)})

store = pd.HDFStore('mydata.h5')
store['obj1'] = frame
store['obj1_col'] = frame['a']

frm2 = store['obj1']
store.close()
```

Reading Microsoft Excel Files

- Pandas supports **reading** from **Excel 2003** (and higher) files using either the **`ExcelFile`** class or **`pandas.read_excel`** function.
- Need the packages **`xlrd`** and **`openpyxl`**.
- Writing is supported with **`ExcelWriter`**.

```
xlsx = pd.ExcelFile('ex1.xlsx')
pd.read_excel(xlsx, 'Sheet1')
   a  b  c  d  message
0  1  2  3  4    hello
1  5  6  7  8    world
2  9 10 11 12      foo
# Alternatively, for one sheet:
frame = pd.read_excel('ex1.xlsx',
                      'Sheet1')

writer = pd.ExcelWriter('ex2.xlsx')
frame.to_excel(writer, 'Sheet1')
writer.save() # Save and close
# Alternatively, for one sheet:
frame.to_excel('ex2.xlsx')
```

Outline

6.1 Reading and Writing Data in Text Format

6.2 Binary Data Formats

6.3 Interacting with Web APIs

6.4 Interacting with Databases

6.3 Interacting with Web APIs

- Many **websites** have public APIs providing data feeds via **JSON**, e.g., Weather Data.

- To find the last **30 GitHub issues** for pandas, we can make a **GET** HTTP request using the add-on **requests** library.

```
import requests
url =
'https://api.github.com/repos/panda
s-dev/pandas/issues'

resp = requests.get(url)
data = resp.json() # list of dict
data[0]['title']
'Period does not round down for …'
issues = pd.DataFrame(data,
          columns=['number',
             'title', 'labels',
             'state'])
```

Outline

6.1 Reading and Writing Data in Text Format

6.2 Binary Data Formats

6.3 Interacting with Web APIs

6.4 Interacting with Databases

6.4 Interacting with Databases

- The SQLAlchemy project is a popular **Python SQL toolkit** for interfacing with SQL databases.
- **Supports** SQLite, Postgresql, MySQL, Oracle, MS-SQL, Firebird, Sybase and others.
- pandas has **`read_sql`** that reads data easily from a SQLAlchemy connection.

```
import sqlalchemy as sqla
db = sqla.create_engine(
          'sqlite:///mydata.sqlite')

pd.read_sql('select * from test',
            db)
```

	a	b	c	d
0	Atlanta	Georgia	1.25	6
1	Tallahassee	Florida	2.60	3
2	Sacramento	California	1.70	5

Homework

- Solve the homework on **data loading and file formats.**

Summary

6.1 Reading and Writing Data in Text Format

6.2 Binary Data Formats

6.3 Interacting with Web APIs

6.4 Interacting with Databases