Data Loading, Storage, and File Formats

Part 4

Binary Data Formats

- One of the easiest ways to store data (also known as *serialization*) efficiently in binary format is using Python's built-in `pickle` serialization.
- pandas objects all have a `to_pickle` method that writes the data to disk in pickle format:

```
In [65]: frame = pd.read_csv('examples/ex1.csv')

In [66]: frame
Out[66]:
```

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

```
In [67]: frame.to_pickle('examples/frame_pickle')
```

- You can read any “pickled” object stored in a file by using the built-in `pickle` directly, or even more conveniently using `pandas.read_pickle`:

```
In [68]: pd.read_pickle('examples/frame_pickle')
Out[68]:
```

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

- `pickle` is only recommended as a short-term storage format.
- The problem is that it is hard to guarantee that the format will be stable over time; an object pickled today may not unpickle with a later version of a library.
- We have tried to maintain backward compatibility when possible, but at some point in the future it may be necessary to “break” the pickle format.

Using HDF5 Format

- HDF5 is a well-regarded file format intended for storing large quantities of scientific array data.
- It is available as a C library, and it has interfaces available in many other languages, including Java, Julia, MATLAB, and Python.
- The “HDF” in HDF5 stands for *hierarchical data format*.
- Each HDF5 file can store multiple datasets and supporting metadata.
- Compared with simpler formats, HDF5 supports on-the-fly compression with a variety of compression modes, enabling data with repeated patterns to be stored more efficiently.
- HDF5 can be a good choice for working with very large datasets that don’t fit into memory, as you can efficiently read and write small sections of much larger arrays.

- While it’s possible to directly access HDF5 files using either the PyTables or h5py libraries, pandas provides a high-level interface that simplifies storing Series and DataFrame object.
- The `HDFStore` class works like a dict and handles the low-level details:

```
In [69]: frame = pd.DataFrame({'a': np.random.randn(100)})

In [70]: store = pd.HDFStore('mydata.h5')

In [71]: store['obj1'] = frame

In [72]: store['obj1_col'] = frame['a']

In [73]: store
Out[73]: <class 'pandas.io.pytables.HDFStore'>
         File path: mydata.h5
```

- Objects contained in the HDF5 file can then be retrieved with the same dict-like API:

```
In [74]: store['obj1']
```

Out[74]:

	a
0	-0.204708
1	0.478943
2	-0.519439
3	-0.555730
4	1.965781
...	...
95	0.795253
96	0.118110
97	-0.748532
98	0.584970
99	0.152677

100 rows × 1 columns

- HDFStore supports two storage schemas, `'fixed'` and `'table'`.
- The latter is generally slower, but it supports query operations using a special syntax:

```
In [75]: store.put('obj2', frame, format='table')

In [76]: store.select('obj2', where=['index >= 10 and index <= 15'])
Out[76]:
```

	a
10	1.007189
11	-1.296221
12	0.274992
13	0.228913
14	1.352917
15	0.886429

```
In [77]: store.close()
```

- The `pandas.read_hdf` function gives you a shortcut to these tools:

```
In [78]: frame.to_hdf('mydata.h5', 'obj3', format='table')

In [79]: pd.read_hdf('mydata.h5', 'obj3', where=['index < 5'])
Out[79]:
```

	a
0	-0.204708
1	0.478943
2	-0.519439
3	-0.555730
4	1.965781

- HDF5 is not a database.
- It is best suited for write-once, read-many datasets.
- While data can be added to a file at any time, if multiple writers do so simultaneously, the file can become corrupted.

Reading Microsoft Excel Files

- pandas also supports reading tabular data stored in Excel 2003 (and higher) files using either the `ExcelFile` class or `pandas.read_excel` function.
- Internally these tools use the add-on packages `xlrd` and `openpyxl` to read XLS and XLSX files, respectively.
- You may need to install these manually with pip or conda.

```
(base) joshua@joshua-Virtual-Machine:~$ conda list | grep xlrd
xlrd                      1.2.0                    py37_0
(base) joshua@joshua-Virtual-Machine:~$ conda list | grep openpyxl
openpyxl                  2.6.1                    py37_1
(base) joshua@joshua-Virtual-Machine:~$
```

- To use `ExcelFile`, create an instance by passing a path to an `xls` or `xlsx` file:

```
In [80]: xlsx = pd.ExcelFile('examples/ex1.xlsx')
```

- Data stored in a sheet can then be read into DataFrame with `parse`:

```
In [81]: pd.read_excel(xlsx, 'Sheet1')
Out[81]:
```

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

- If you are reading multiple sheets in a file, then it is faster to create the `ExcelFile`, but you can also simply pass the filename to `pandas.read_excel`:

```
In [82]: frame = pd.read_excel('examples/ex1.xlsx', 'Sheet1')

In [83]: frame
Out[83]:
```

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

- To write pandas data to Excel format, you must first create an `ExcelWriter`, then write data to it using pandas objects' `to_excel` method:

```
In [84]: writer = pd.ExcelWriter('examples/ex2.xlsx')

In [85]: frame.to_excel(writer, 'Sheet1')

In [86]: writer.save()
```

- You can also pass a file path to `to_excel` and avoid the `ExcelWriter`:

```
In [87]: frame.to_excel('examples/ex2.xlsx')
```