

# Data Loading, Storage, and File Formats

The tools in this book are of little use if you can't easily import and export data in Python. I'm going to be focused on input and output with pandas objects, though there are of course numerous tools in other libraries to aid in this process. NumPy, for example, features low-level but extremely fast binary data loading and storage, including support for memory-mapped array. See [Chapter 12](#) for more on those.

Input and output typically falls into a few main categories: reading text files and other more efficient on-disk formats, loading data from databases, and interacting with network sources like web APIs.

## Reading and Writing Data in Text Format

Python has become a beloved language for text and file munging due to its simple syntax for interacting with files, intuitive data structures, and convenient features like tuple packing and unpacking.

pandas features a number of functions for reading tabular data as a DataFrame object. [Table 6-1](#) has a summary of all of them, though `read_csv` and `read_table` are likely the ones you'll use the most.

*Table 6-1. Parsing functions in pandas*

Function	Description
<code>read_csv</code>	Load delimited data from a file, URL, or file-like object. Use comma as default delimiter
<code>read_table</code>	Load delimited data from a file, URL, or file-like object. Use tab (' <code>\t</code> ') as default delimiter
<code>read_fwf</code>	Read data in fixed-width column format (that is, no delimiters)
<code>read_clipboard</code>	Version of <code>read_table</code> that reads data from the clipboard. Useful for converting tables from web pages

I'll give an overview of the mechanics of these functions, which are meant to convert text data into a DataFrame. The options for these functions fall into a few categories:

- Indexing: can treat one or more columns as the returned DataFrame, and whether to get column names from the file, the user, or not at all.
- Type inference and data conversion: this includes the user-defined value conversions and custom list of missing value markers.
- Datetime parsing: includes combining capability, including combining date and time information spread over multiple columns into a single column in the result.
- Iterating: support for iterating over chunks of very large files.
- Unclean data issues: skipping rows or a footer, comments, or other minor things like numeric data with thousands separated by commas.

*Type inference* is one of the more important features of these functions; that means you don't have to specify which columns are numeric, integer, boolean, or string. Handling dates and other custom types requires a bit more effort, though. Let's start with a small comma-separated (CSV) text file:

```
In [846]: !cat ch06/ex1.csv
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

Since this is comma-delimited, we can use `read_csv` to read it into a DataFrame:

```
In [847]: df = pd.read_csv('ch06/ex1.csv')

In [848]: df
Out[848]:
```

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

We could also have used `read_table` and specifying the delimiter:

```
In [849]: pd.read_table('ch06/ex1.csv', sep=',')
Out[849]:
```

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



Here I used the Unix `cat` shell command to print the raw contents of the file to the screen. If you're on Windows, you can use `type` instead of `cat` to achieve the same effect.

A file will not always have a header row. Consider this file:

```
In [850]: !cat ch06/ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

To read this in, you have a couple of options. You can allow pandas to assign default column names, or you can specify names yourself:

```
In [851]: pd.read_csv('ch06/ex2.csv', header=None)
Out[851]:
```

	X.1	X.2	X.3	X.4	X.5
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

```
In [852]: pd.read_csv('ch06/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
Out[852]:
```

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

Suppose you wanted the `message` column to be the index of the returned DataFrame. You can either indicate you want the column at index 4 or named `'message'` using the `index_col` argument:

```
In [853]: names = ['a', 'b', 'c', 'd', 'message']

In [854]: pd.read_csv('ch06/ex2.csv', names=names, index_col='message')
Out[854]:
```

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

In the event that you want to form a hierarchical index from multiple columns, just pass a list of column numbers or names:

```
In [855]: !cat ch06/csv_mindex.csv
key1,key2,value1,value2
one,a,1,2
one,b,3,4
one,c,5,6
one,d,7,8
two,a,9,10
two,b,11,12
two,c,13,14
two,d,15,16

In [856]: parsed = pd.read_csv('ch06/csv_mindex.csv', index_col=['key1', 'key2'])

In [857]: parsed
Out[857]:
```

		value1	value2
key1	key2		
one	a	1	2
	b	3	4
	c	5	6
	d	7	8
two	a	9	10
	b	11	12
	c	13	14
	d	15	16

In some cases, a table might not have a fixed delimiter, using whitespace or some other pattern to separate fields. In these cases, you can pass a regular expression as a delimiter for `read_table`. Consider a text file that looks like this:

```
In [858]: list(open('ch06/ex3.txt'))
Out[858]:
['      A      B      C\n',
'aaa -0.264438 -1.026059 -0.619500\n',
'bbb  0.927272  0.302904 -0.032399\n',
'ccc -0.264273 -0.386314 -0.217601\n',
'ddd -0.871858 -0.348382  1.100491\n']
```

While you could do some munging by hand, in this case fields are separated by a variable amount of whitespace. This can be expressed by the regular expression `\s+`, so we have then:

```
In [859]: result = pd.read_table('ch06/ex3.txt', sep='\s+')

In [860]: result
Out[860]:
      A      B      C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382  1.100491
```

Because there was one fewer column name than the number of data rows, `read_table` infers that the first column should be the DataFrame's index in this special case.

The parser functions have many additional arguments to help you handle the wide variety of exception file formats that occur (see [Table 6-2](#)). For example, you can skip the first, third, and fourth rows of a file with `skiprows`:

```
In [861]: !cat ch06/ex4.csv
# hey!
a,b,c,d,message
# just wanted to make things more difficult for you
# who reads CSV files with computers, anyway?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
In [862]: pd.read_csv('ch06/ex4.csv', skiprows=[0, 2, 3])
Out[862]:
   a    b    c    d message
```

```

0  1  2  3  4  hello
1  5  6  7  8  world
2  9 10 11 12   foo

```

Handling missing values is an important and frequently nuanced part of the file parsing process. Missing data is usually either not present (empty string) or marked by some *sentinel* value. By default, pandas uses a set of commonly occurring sentinels, such as NA, -1.#IND, and NULL:

```

In [863]: !cat ch06/ex5.csv
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo
In [864]: result = pd.read_csv('ch06/ex5.csv')

```

```

In [865]: result
Out[865]:
  something  a  b  c  d message
0      one  1  2  3  4     NaN
1      two  5  6 NaN  8    world
2     three  9 10 11 12     foo

```

```

In [866]: pd.isnull(result)
Out[866]:
  something  a  b  c  d message
0     False False False False False   True
1     False False False  True False  False
2     False False False False False  False

```

The `na_values` option can take either a list or set of strings to consider missing values:

```

In [867]: result = pd.read_csv('ch06/ex5.csv', na_values=['NULL'])

```

```

In [868]: result
Out[868]:
  something  a  b  c  d message
0      one  1  2  3  4     NaN
1      two  5  6 NaN  8    world
2     three  9 10 11 12     foo

```

Different NA sentinels can be specified for each column in a dict:

```

In [869]: sentinels = {'message': ['foo', 'NA'], 'something': ['two']}

```

```

In [870]: pd.read_csv('ch06/ex5.csv', na_values=sentinels)
Out[870]:
  something  a  b  c  d message
0      one  1  2  3  4     NaN
1      NaN  5  6 NaN  8    world
2     three  9 10 11 12     NaN

```

Table 6-2. *read\_csv/read\_table* function arguments

Argument	Description
path	String indicating filesystem location, URL, or file-like object
sep or delimiter	Character sequence or regular expression to use to split fields in each row
header	Row number to use as column names. Defaults to 0 (first row), but should be None if there is no header row
index_col	Column numbers or names to use as the row index in the result. Can be a single name/number or a list of them for a hierarchical index
names	List of column names for result, combine with header=None
skiprows	Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip
na_values	Sequence of values to replace with NA
comment	Character or characters to split comments off the end of lines
parse_dates	Attempt to parse data to datetime; False by default. If True, will attempt to parse all columns. Otherwise can specify a list of column numbers or name to parse. If element of list is tuple or list, will combine multiple columns together and parse to date (for example if date/time split across two columns)
keep_date_col	If joining columns to parse date, keep the joined columns. Default False
converters	Dict containing column number of name mapping to functions. For example { 'foo' : f } would apply the function f to all values in the 'foo' column
dayfirst	When parsing potentially ambiguous dates, treat as international format (e.g. 7/6/2012 -> June 7, 2012). Default False
date_parser	Function to use to parse dates
nrows	Number of rows to read from beginning of file
iterator	Return a TextParser object for reading file piecemeal
chunksize	For iteration, size of file chunks
skip_footer	Number of lines to ignore at end of file
verbose	Print various parser output information, like the number of missing values placed in non-numeric columns
encoding	Text encoding for unicode. For example 'utf-8' for UTF-8 encoded text
squeeze	If the parsed data only contains one column return a Series
thousands	Separator for thousands, e.g. ',' or '.'

## Reading Text Files in Pieces

When processing very large files or figuring out the right set of arguments to correctly process a large file, you may only want to read in a small piece of a file or iterate through smaller chunks of the file.

```
In [871]: result = pd.read_csv('ch06/ex6.csv')
```

```
In [872]: result
Out[872]:
```

```

<class 'pandas.core.frame.DataFrame'>
Int64Index: 10000 entries, 0 to 9999
Data columns:
one      10000  non-null values
two      10000  non-null values
three    10000  non-null values
four     10000  non-null values
key      10000  non-null values
dtypes: float64(4), object(1)

```

If you want to only read out a small number of rows (avoiding reading the entire file), specify that with `nrows`:

```

In [873]: pd.read_csv('ch06/ex6.csv', nrows=5)
Out[873]:
   one      two      three      four key
0  0.467976 -0.038649 -0.295344 -1.824726 L
1 -0.358893  1.404453  0.704965 -0.200638 B
2 -0.501840  0.659254 -0.421691 -0.057688 G
3  0.204886  1.074134  1.388361 -0.982404 R
4  0.354628 -0.133116  0.283763 -0.837063 Q

```

To read out a file in pieces, specify a `chunksize` as a number of rows:

```

In [874]: chunker = pd.read_csv('ch06/ex6.csv', chunksize=1000)

In [875]: chunker
Out[875]: <pandas.io.parsers.TextParser at 0x8398150>

```

The `TextParser` object returned by `read_csv` allows you to iterate over the parts of the file according to the `chunksize`. For example, we can iterate over `ex6.csv`, aggregating the value counts in the 'key' column like so:

```

chunker = pd.read_csv('ch06/ex6.csv', chunksize=1000)

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

tot = tot.order(ascending=False)

```

We have then:

```

In [877]: tot[:10]
Out[877]:
E    368
X    364
L    346
O    343
Q    340
M    338
J    337
F    335
K    334
H    330

```

TextParser is also equipped with a `get_chunk` method which enables you to read pieces of an arbitrary size.

## Writing Data Out to Text Format

Data can also be exported to delimited format. Let's consider one of the CSV files read above:

```
In [878]: data = pd.read_csv('ch06/ex5.csv')
```

```
In [879]: data
```

```
Out[879]:
```

```
  something  a  b  c  d message
0      one  1  2  3  4      NaN
1      two  5  6 NaN  8    world
2     three  9 10 11 12     foo
```

Using DataFrame's `to_csv` method, we can write the data out to a comma-separated file:

```
In [880]: data.to_csv('ch06/out.csv')
```

```
In [881]: !cat ch06/out.csv
```

```
,something,a,b,c,d,message
0,one,1,2,3.0,4,
1,two,5,6,,8,world
2,three,9,10,11.0,12,foo
```

Other delimiters can be used, of course (writing to `sys.stdout` so it just prints the text result):

```
In [882]: data.to_csv(sys.stdout, sep='|')
|something|a|b|c|d|message
0|one|1|2|3.0|4|
1|two|5|6|8|world
2|three|9|10|11.0|12|foo
```

Missing values appear as empty strings in the output. You might want to denote them by some other sentinel value:

```
In [883]: data.to_csv(sys.stdout, na_rep='NULL')
,something,a,b,c,d,message
0,one,1,2,3.0,4,NULL
1,two,5,6,NULL,8,world
2,three,9,10,11.0,12,foo
```

With no other options specified, both the row and column labels are written. Both of these can be disabled:

```
In [884]: data.to_csv(sys.stdout, index=False, header=False)
one,1,2,3.0,4,
two,5,6,,8,world
three,9,10,11.0,12,foo
```

You can also write only a subset of the columns, and in an order of your choosing:



```
In [885]: data.to_csv(sys.stdout, index=False, cols=['a', 'b', 'c'])
a,b,c
1,2,3.0
5,6,
9,10,11.0
```

Series also has a `to_csv` method:

```
In [886]: dates = pd.date_range('1/1/2000', periods=7)

In [887]: ts = Series(np.arange(7), index=dates)

In [888]: ts.to_csv('ch06/tseries.csv')

In [889]: !cat ch06/tseries.csv
2000-01-01 00:00:00,0
2000-01-02 00:00:00,1
2000-01-03 00:00:00,2
2000-01-04 00:00:00,3
2000-01-05 00:00:00,4
2000-01-06 00:00:00,5
2000-01-07 00:00:00,6
```

With a bit of wrangling (no header, first column as index), you can read a CSV version of a Series with `read_csv`, but there is also a `from_csv` convenience method that makes it a bit simpler:

```
In [890]: Series.from_csv('ch06/tseries.csv', parse_dates=True)
Out[890]:
2000-01-01    0
2000-01-02    1
2000-01-03    2
2000-01-04    3
2000-01-05    4
2000-01-06    5
2000-01-07    6
```

See the docstrings for `to_csv` and `from_csv` in IPython for more information.

## Manually Working with Delimited Formats

Most forms of tabular data can be loaded from disk using functions like  `pandas.read_table`. In some cases, however, some manual processing may be necessary. It's not uncommon to receive a file with one or more malformed lines that trip up `read_table`. To illustrate the basic tools, consider a small CSV file:

```
In [891]: !cat ch06/ex7.csv
"a","b","c"
"1","2","3"
"1","2","3","4"
```

For any file with a single-character delimiter, you can use Python's built-in `csv` module. To use it, pass any open file or file-like object to `csv.reader`:

```
import csv
f = open('ch06/ex7.csv')
```

```
reader = csv.reader(f)
```

Iterating through the reader like a file yields tuples of values in each like with any quote characters removed:

```
In [893]: for line in reader:
.....:     print line
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3', '4']
```

From there, it's up to you to do the wrangling necessary to put the data in the form that you need it. For example:

```
In [894]: lines = list(csv.reader(open('ch06/ex7.csv')))

In [895]: header, values = lines[0], lines[1:]

In [896]: data_dict = {h: v for h, v in zip(header, zip(*values))}

In [897]: data_dict
Out[897]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

CSV files come in many different flavors. Defining a new format with a different delimiter, string quoting convention, or line terminator is done by defining a simple subclass of `csv.Dialect`:

```
class my_dialect(csv.Dialect):
    lineterminator = '\n'
    delimiter = ';'
    quotechar = '"'

reader = csv.reader(f, dialect=my_dialect)
```

Individual CSV dialect parameters can also be given as keywords to `csv.reader` without having to define a subclass:

```
reader = csv.reader(f, delimiter='|')
```

The possible options (attributes of `csv.Dialect`) and what they do can be found in [Table 6-3](#).

*Table 6-3. CSV dialect options*

Argument	Description
<code>delimiter</code>	One-character string to separate fields. Defaults to <code>,</code> .
<code>lineterminator</code>	Line terminator for writing, defaults to <code>\r\n</code> . Reader ignores this and recognizes cross-platform line terminators.
<code>quotechar</code>	Quote character for fields with special characters (like a delimiter). Default is <code>"</code> .
<code>quoting</code>	Quoting convention. Options include <code>csv.QUOTE_ALL</code> (quote all fields), <code>csv.QUOTE_MINIMAL</code> (only fields with special characters like the delimiter),

Argument	Description
	<code>csv.QUOTE_NONNUMERIC</code> , and <code>csv.QUOTE_NON</code> (no quoting). See Python's documentation for full details. Defaults to <code>QUOTE_MINIMAL</code> .
<code>skipinitialspace</code>	Ignore whitespace after each delimiter. Default <code>False</code> .
<code>doublequote</code>	How to handle quoting character inside a field. If <code>True</code> , it is doubled. See online documentation for full detail and behavior.
<code>escapechar</code>	String to escape the delimiter if quoting is set to <code>csv.QUOTE_NONE</code> . Disabled by default



For files with more complicated or fixed multicharacter delimiters, you will not be able to use the `csv` module. In those cases, you'll have to do the line splitting and other cleanup using string's `split` method or the regular expression method `re.split`.

To *write* delimited files manually, you can use `csv.writer`. It accepts an open, writable file object and the same dialect and format options as `csv.reader`:

```
with open('mydata.csv', 'w') as f:
    writer = csv.writer(f, dialect=my_dialect)
    writer.writerow(('one', 'two', 'three'))
    writer.writerow(('1', '2', '3'))
    writer.writerow(('4', '5', '6'))
    writer.writerow(('7', '8', '9'))
```

## JSON Data

JSON (short for JavaScript Object Notation) has become one of the standard formats for sending data by HTTP request between web browsers and other applications. It is a much more flexible data format than a tabular text form like CSV. Here is an example:

```
obj = """
{"name": "Wes",
 "places_lived": ["United States", "Spain", "Germany"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 25, "pet": "Zuko"},
               {"name": "Katie", "age": 33, "pet": "Cisco"}]}
"""
```

JSON is very nearly valid Python code with the exception of its null value `null` and some other nuances (such as disallowing trailing commas at the end of lists). The basic types are objects (dicts), arrays (lists), strings, numbers, booleans, and nulls. All of the keys in an object must be strings. There are several Python libraries for reading and writing JSON data. I'll use `json` here as it is built into the Python standard library. To convert a JSON string to Python form, use `json.loads`:

```
In [899]: import json
```

```
In [900]: result = json.loads(obj)

In [901]: result
Out[901]:
{'u'name': u'Wes',
 u'pet': None,
 u'places_lived': [u'United States', u'Spain', u'Germany'],
 u'siblings': [{u'age': 25, u'name': u'Scott', u'pet': u'Zuko'},
 {u'age': 33, u'name': u'Katie', u'pet': u'Cisco'}]}
```

`json.dumps` on the other hand converts a Python object back to JSON:

```
In [902]: asjson = json.dumps(result)
```

How you convert a JSON object or list of objects to a DataFrame or some other data structure for analysis will be up to you. Conveniently, you can pass a list of JSON objects to the DataFrame constructor and select a subset of the data fields:

```
In [903]: siblings = DataFrame(result['siblings'], columns=['name', 'age'])

In [904]: siblings
Out[904]:
   name  age
0  Scott   25
1  Katie   33
```

For an extended example of reading and manipulating JSON data (including nested records), see the USDA Food Database example in the next chapter.



An effort is underway to add fast native JSON export (`to_json`) and decoding (`from_json`) to pandas. This was not ready at the time of writing.

## XML and HTML: Web Scraping

Python has many libraries for reading and writing data in the ubiquitous HTML and XML formats. `lxml` (<http://lxml.de>) is one that has consistently strong performance in parsing very large files. `lxml` has multiple programmer interfaces; first I'll show using `lxml.html` for HTML, then parse some XML using `lxml.objectify`.

Many websites make data available in HTML tables for viewing in a browser, but not downloadable as an easily machine-readable format like JSON, HTML, or XML. I noticed that this was the case with Yahoo! Finance's stock options data. If you aren't familiar with this data; options are derivative contracts giving you the right to buy (*call* option) or sell (*put* option) a company's stock at some particular price (the *strike*) between now and some fixed point in the future (the *expiry*). People trade both *call* and *put* options across many strikes and expiries; this data can all be found together in tables on Yahoo! Finance.

To get started, find the URL you want to extract data from, open it with `urllib2` and parse the stream with `lxml` like so:

```
from lxml.html import parse
from urllib2 import urlopen

parsed = parse(urlopen('http://finance.yahoo.com/q/op?s=AAPL+Options'))

doc = parsed.getroot()
```

Using this object, you can extract all HTML tags of a particular type, such as `table` tags containing the data of interest. As a simple motivating example, suppose you wanted to get a list of every URL linked to in the document; links are `a` tags in HTML. Using the document root's `findall` method along with an XPath (a means of expressing “queries” on the document):

```
In [906]: links = doc.findall('..//a')
```

```
In [907]: links[15:20]
Out[907]:
[<Element a at 0x6c488f0>,
 <Element a at 0x6c48950>,
 <Element a at 0x6c489b0>,
 <Element a at 0x6c48a10>,
 <Element a at 0x6c48a70>]
```

But these are objects representing HTML elements; to get the URL and link text you have to use each element's `get` method (for the URL) and `text_content` method (for the display text):

```
In [908]: lnk = links[28]

In [909]: lnk
Out[909]: <Element a at 0x6c48dd0>

In [910]: lnk.get('href')
Out[910]: 'http://biz.yahoo.com/special.html'

In [911]: lnk.text_content()
Out[911]: 'Special Editions'
```

Thus, getting a list of all URLs in the document is a matter of writing this list comprehension:

```
In [912]: urls = [lnk.get('href') for lnk in doc.findall('..//a')]

In [913]: urls[-10:]
Out[913]:
['http://info.yahoo.com/privacy/us/yahoo/finance/details.html',
 'http://info.yahoo.com/relevantads/',
 'http://docs.yahoo.com/info/terms/',
 'http://docs.yahoo.com/info/copyright/copyright.html',
 'http://help.yahoo.com/l/us/yahoo/finance/forms_index.html',
 'http://help.yahoo.com/l/us/yahoo/finance/quotes/fitadelay.html',
 'http://help.yahoo.com/l/us/yahoo/finance/quotes/fitadelay.html',
```

```
'http://www.capitaliq.com',
'http://www.csidata.com',
'http://www.morningstar.com/']
```

Now, finding the right tables in the document can be a matter of trial and error; some websites make it easier by giving a table of interest an `id` attribute. I determined that these were the two tables containing the call data and put data, respectively:

```
tables = doc.findall('.//table')
calls = tables[9]
puts = tables[13]
```

Each table has a header row followed by each of the data rows:

```
In [915]: rows = calls.findall('.//tr')
```

For the header as well as the data rows, we want to extract the text from each cell; in the case of the header these are `th` cells and `td` cells for the data:

```
def _unpack(row, kind='td'):
    elts = row.findall('.//%s' % kind)
    return [val.text_content() for val in elts]
```

Thus, we obtain:

```
In [917]: _unpack(rows[0], kind='th')
Out[917]: ['Strike', 'Symbol', 'Last', 'Chg', 'Bid', 'Ask', 'Vol', 'Open Int']

In [918]: _unpack(rows[1], kind='td')
Out[918]:
['295.00',
'AAPL120818C00295000',
'310.40',
'0.00',
'289.80',
'290.80',
'1',
'169']
```

Now, it's a matter of combining all of these steps together to convert this data into a `DataFrame`. Since the numerical data is still in string format, we want to convert some, but perhaps not all of the columns to floating point format. You could do this by hand, but, luckily, `pandas` has a class `TextParser` that is used internally in the `read_csv` and other parsing functions to do the appropriate automatic type conversion:

```
from pandas.io.parsers import TextParser

def parse_options_data(table):
    rows = table.findall('.//tr')
    header = _unpack(rows[0], kind='th')
    data = [_unpack(r) for r in rows[1:]]
    return TextParser(data, names=header).get_chunk()
```

Finally, we invoke this parsing function on the `lxml` table objects and get `DataFrame` results:

```
In [920]: call_data = parse_options_data(calls)
```

```
In [921]: put_data = parse_options_data(puts)
```

```
In [922]: call_data[:10]
```

```
Out[922]:
```

	Strike	Symbol	Last	Chg	Bid	Ask	Vol	Open	Int
0	295	AAPL120818C00295000	310.40	0.0	289.80	290.80	1		169
1	300	AAPL120818C00300000	277.10	1.7	284.80	285.60	2		478
2	305	AAPL120818C00305000	300.97	0.0	279.80	280.80	10		316
3	310	AAPL120818C00310000	267.05	0.0	274.80	275.65	6		239
4	315	AAPL120818C00315000	296.54	0.0	269.80	270.80	22		88
5	320	AAPL120818C00320000	291.63	0.0	264.80	265.80	96		173
6	325	AAPL120818C00325000	261.34	0.0	259.80	260.80	N/A		108
7	330	AAPL120818C00330000	230.25	0.0	254.80	255.80	N/A		21
8	335	AAPL120818C00335000	266.03	0.0	249.80	250.65	4		46
9	340	AAPL120818C00340000	272.58	0.0	244.80	245.80	4		30

## Parsing XML with lxml.objectify

XML (extensible markup language) is another common structured data format supporting hierarchical, nested data with metadata. The files that generate the book you are reading actually form a series of large XML documents.

Above, I showed the lxml library and its lxml.html interface. Here I show an alternate interface that's convenient for XML data, lxml.objectify.

The New York Metropolitan Transportation Authority (MTA) publishes a number of data series about its bus and train services (<http://www.mta.info/developers/download.html>). Here we'll look at the performance data which is contained in a set of XML files. Each train or bus service has a different file (like Performance\_MNR.xml for the Metro-North Railroad) containing monthly data as a series of XML records that look like this:

```
<INDICATOR>
  <INDICATOR_SEQ>373889</INDICATOR_SEQ>
  <PARENT_SEQ></PARENT_SEQ>
  <AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>
  <INDICATOR_NAME>Escalator Availability</INDICATOR_NAME>
  <DESCRIPTION>Percent of the time that escalators are operational
systemwide. The availability rate is based on physical observations performed
the morning of regular business days only. This is a new indicator the agency
began reporting in 2009.</DESCRIPTION>
  <PERIOD_YEAR>2011</PERIOD_YEAR>
  <PERIOD_MONTH>12</PERIOD_MONTH>
  <CATEGORY>Service Indicators</CATEGORY>
  <FREQUENCY>M</FREQUENCY>
  <DESIRED_CHANGE>U</DESIRED_CHANGE>
  <INDICATOR_UNIT>%</INDICATOR_UNIT>
  <DECIMAL_PLACES>1</DECIMAL_PLACES>
  <YTD_TARGET>97.00</YTD_TARGET>
  <YTD_ACTUAL></YTD_ACTUAL>
  <MONTHLY_TARGET>97.00</MONTHLY_TARGET>
  <MONTHLY_ACTUAL></MONTHLY_ACTUAL>
</INDICATOR>
```

Using `lxml.objectify`, we parse the file and get a reference to the root node of the XML file with `getroot`:

```
from lxml import objectify

path = 'Performance_MNR.xml'
parsed = objectify.parse(open(path))
root = parsed.getroot()
```

`root.INDICATOR` return a generator yielding each `<INDICATOR>` XML element. For each record, we can populate a dict of tag names (like `YTD_ACTUAL`) to data values (excluding a few tags):

```
data = []

skip_fields = ['PARENT_SEQ', 'INDICATOR_SEQ',
               'DESIRED_CHANGE', 'DECIMAL_PLACES']

for elt in root.INDICATOR:
    el_data = {}
    for child in elt.getchildren():
        if child.tag in skip_fields:
            continue
        el_data[child.tag] = child.pyval
    data.append(el_data)
```

Lastly, convert this list of dicts into a `DataFrame`:

```
In [927]: perf = DataFrame(data)

In [928]: perf
Out[928]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 648 entries, 0 to 647
Data columns:
AGENCY_NAME      648 non-null values
CATEGORY         648 non-null values
DESCRIPTION       648 non-null values
FREQUENCY        648 non-null values
INDICATOR_NAME   648 non-null values
INDICATOR_UNIT   648 non-null values
MONTHLY_ACTUAL   648 non-null values
MONTHLY_TARGET   648 non-null values
PERIOD_MONTH     648 non-null values
PERIOD_YEAR      648 non-null values
YTD_ACTUAL       648 non-null values
YTD_TARGET       648 non-null values
dtypes: int64(2), object(10)
```

XML data can get much more complicated than this example. Each tag can have metadata, too. Consider an HTML link tag which is also valid XML:

```
from StringIO import StringIO
tag = '<a href="http://www.google.com">Google</a>'

root = objectify.parse(StringIO(tag)).getroot()
```



You can now access any of the fields (like `href`) in the tag or the link text:

```
In [930]: root
Out[930]: <Element a at 0x88bd4b0>
```

```
In [931]: root.get('href')
Out[931]: 'http://www.google.com'
```

```
In [932]: root.text
Out[932]: 'Google'
```

## Binary Data Formats

One of the easiest ways to store data efficiently in binary format is using Python’s built-in `pickle` serialization. Conveniently, pandas objects all have a `save` method which writes the data to disk as a pickle:

```
In [933]: frame = pd.read_csv('ch06/ex1.csv')
```

```
In [934]: frame
Out[934]:
```

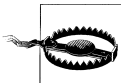
	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

```
In [935]: frame.save('ch06/frame_pickle')
```

You read the data back into Python with `pandas.load`, another pickle convenience function:

```
In [936]: pd.load('ch06/frame_pickle')
Out[936]:
```

	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. I have made every effort to ensure that this does not occur with pandas, but at some point in the future it may be necessary to “break” the pickle format.

## Using HDF5 Format

There are a number of tools that facilitate efficiently reading and writing large amounts of scientific data in binary format on disk. A popular industry-grade library for this is HDF5, which is a C library with interfaces in many other languages like Java, Python, and MATLAB. The “HDF” in HDF5 stands for *hierarchical data format*. Each HDF5

file contains an internal file system-like node structure enabling you to store multiple datasets and supporting metadata. Compared with simpler formats, HDF5 supports on-the-fly compression with a variety of compressors, enabling data with repeated patterns to be stored more efficiently. For very large datasets that don't fit into memory, HDF5 is a good choice as you can efficiently read and write small sections of much larger arrays.

There are not one but two interfaces to the HDF5 library in Python, PyTables and h5py, each of which takes a different approach to the problem. h5py provides a direct, but high-level interface to the HDF5 API, while PyTables abstracts many of the details of HDF5 to provide multiple flexible data containers, table indexing, querying capability, and some support for out-of-core computations.

pandas has a minimal dict-like `HDFStore` class, which uses PyTables to store pandas objects:

```
In [937]: store = pd.HDFStore('mydata.h5')
```

```
In [938]: store['obj1'] = frame
```

```
In [939]: store['obj1_col'] = frame['a']
```

```
In [940]: store
```

```
Out[940]:
```

```
<class 'pandas.io.pytables.HDFStore'>
```

```
File path: mydata.h5
```

```
obj1      DataFrame
```

```
obj1_col  Series
```

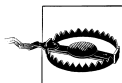
Objects contained in the HDF5 file can be retrieved in a dict-like fashion:

```
In [941]: store['obj1']
```

```
Out[941]:
```

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

If you work with huge quantities of data, I would encourage you to explore PyTables and h5py to see how they can suit your needs. Since many data analysis problems are IO-bound (rather than CPU-bound), using a tool like HDF5 can massively accelerate your applications.



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 the `ExcelFile` class. Internally `ExcelFile` uses the `xlrd` and `openpyxl` packages, so you

may have to install them first. To use `ExcelFile`, create an instance by passing a path to an `xls` or `xlsx` file:

```
xls_file = pd.ExcelFile('data.xls')
```

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

```
table = xls_file.parse('Sheet1')
```

## Interacting with HTML and Web APIs

Many websites have public APIs providing data feeds via JSON or some other format. There are a number of ways to access these APIs from Python; one easy-to-use method that I recommend is the `requests` package (<http://docs.python-requests.org>). To search for the words “python pandas” on Twitter, we can make an HTTP GET request like so:

```
In [944]: import requests
```

```
In [945]: url = 'http://search.twitter.com/search.json?q=python%20pandas'
```

```
In [946]: resp = requests.get(url)
```

```
In [947]: resp
```

```
Out[947]: <Response [200]>
```

The `Response` object’s `text` attribute contains the content of the GET query. Many web APIs will return a JSON string that must be loaded into a Python object:

```
In [948]: import json
```

```
In [949]: data = json.loads(resp.text)
```

```
In [950]: data.keys()
```

```
Out[950]:
```

```
[u'next_page',  
 u'completed_in',  
 u'max_id_str',  
 u'since_id_str',  
 u'refresh_url',  
 u'results',  
 u'since_id',  
 u'results_per_page',  
 u'query',  
 u'max_id',  
 u'page']
```

The `results` field in the response contains a list of tweets, each of which is represented as a Python dict that looks like:

```
{u'created_at': u'Mon, 25 Jun 2012 17:50:33 +0000',  
 u'from_user': u'wesmckinn',  
 u'from_user_id': 115494880,  
 u'from_user_id_str': u'115494880',  
 u'from_user_name': u'Wes McKinney',  
 u'geo': None,
```

```

u'id': 217313849177686018,
u'id_str': u'217313849177686018',
u'iso_language_code': u'pt',
u'metadata': {u'result_type': u'recent'},
u'source': u'<a href="http://twitter.com/">web</a>',
u'text': u'Lunchtime pandas-fu http://t.co/SI70xZZQ #pydata',
u'to_user': None,
u'to_user_id': 0,
u'to_user_id_str': u'0',
u'to_user_name': None}

```

We can then make a list of the tweet fields of interest then pass the results list to DataFrame:

```

In [951]: tweet_fields = ['created_at', 'from_user', 'id', 'text']

In [952]: tweets = DataFrame(data['results'], columns=tweet_fields)

In [953]: tweets
Out[953]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 15 entries, 0 to 14
Data columns:
created_at    15 non-null values
from_user     15 non-null values
id            15 non-null values
text          15 non-null values
dtypes: int64(1), object(3)

```

Each row in the DataFrame now has the extracted data from each tweet:

```

In [121]: tweets.ix[7]
Out[121]:
created_at    Thu, 23 Jul 2012 09:54:00 +0000
from_user      deblike
id            227419585803059201
text          pandas: powerful Python data analysis toolkit
Name: 7

```

With a bit of elbow grease, you can create some higher-level interfaces to common web APIs that return DataFrame objects for easy analysis.

## Interacting with Databases

In many applications data rarely comes from text files, that being a fairly inefficient way to store large amounts of data. SQL-based relational databases (such as SQL Server, PostgreSQL, and MySQL) are in wide use, and many alternative non-SQL (so-called *NoSQL*) databases have become quite popular. The choice of database is usually dependent on the performance, data integrity, and scalability needs of an application.

Loading data from SQL into a DataFrame is fairly straightforward, and pandas has some functions to simplify the process. As an example, I'll use an in-memory SQLite database using Python's built-in `sqlite3` driver:

```
import sqlite3

query = """
CREATE TABLE test
(a VARCHAR(20), b VARCHAR(20),
 c REAL,        d INTEGER
);"""

con = sqlite3.connect(':memory:')
con.execute(query)
con.commit()
```

Then, insert a few rows of data:

```
data = [('Atlanta', 'Georgia', 1.25, 6),
        ('Tallahassee', 'Florida', 2.6, 3),
        ('Sacramento', 'California', 1.7, 5)]
stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

con.executemany(stmt, data)
con.commit()
```

Most Python SQL drivers (PyODBC, pycpg2, MySQLdb, pymssql, etc.) return a list of tuples when selecting data from a table:

```
In [956]: cursor = con.execute('select * from test')
```

```
In [957]: rows = cursor.fetchall()
```

```
In [958]: rows
Out[958]:
[(u'Atlanta', u'Georgia', 1.25, 6),
 (u'Tallahassee', u'Florida', 2.6, 3),
 (u'Sacramento', u'California', 1.7, 5)]
```

You can pass the list of tuples to the DataFrame constructor, but you also need the column names, contained in the cursor's `description` attribute:

```
In [959]: cursor.description
Out[959]:
 (('a', None, None, None, None, None),
 ('b', None, None, None, None, None),
 ('c', None, None, None, None, None),
 ('d', None, None, None, None, None))

In [960]: DataFrame(rows, columns=zip(*cursor.description)[0])
Out[960]:
```

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

This is quite a bit of munging that you'd rather not repeat each time you query the database. pandas has a `read_frame` function in its `pandas.io.sql` module that simplifies the process. Just pass the select statement and the connection object:

```
In [961]: import pandas.io.sql as sql

In [962]: sql.read_frame('select * from test', con)
Out[962]:
```

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

## Storing and Loading Data in MongoDB

NoSQL databases take many different forms. Some are simple dict-like key-value stores like BerkeleyDB or Tokyo Cabinet, while others are document-based, with a dict-like object being the basic unit of storage. I've chosen MongoDB (<http://mongodb.org>) for my example. I started a MongoDB instance locally on my machine, and connect to it on the default port using `pymongo`, the official driver for MongoDB:

```
import pymongo
con = pymongo.Connection('localhost', port=27017)
```

Documents stored in MongoDB are found in collections inside databases. Each running instance of the MongoDB server can have multiple databases, and each database can have multiple collections. Suppose I wanted to store the Twitter API data from earlier in the chapter. First, I can access the (currently empty) `tweets` collection:

```
tweets = con.db.tweets
```

Then, I load the list of tweets and write each of them to the collection using `tweets.save` (which writes the Python dict to MongoDB):

```
import requests, json
url = 'http://search.twitter.com/search.json?q=python%20pandas'
data = json.loads(requests.get(url).text)

for tweet in data['results']:
    tweets.save(tweet)
```

Now, if I wanted to get all of my tweets (if any) from the collection, I can query the collection with the following syntax:

```
cursor = tweets.find({'from_user': 'wesmckinn'})
```

The cursor returned is an iterator that yields each document as a dict. As above I can convert this into a `DataFrame`, optionally extracting a subset of the data fields in each tweet:

```
tweet_fields = ['created_at', 'from_user', 'id', 'text']
result = DataFrame(list(cursor), columns=tweet_fields)
```