CHAPTER 6

Data Loading, Storage, and File Formats

Accessing data is a necessary first step for using most of the tools in this book. I'm going to be focused on data input and output using pandas, though there are numerous tools in other libraries to help with reading and writing data in various formats.

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.

6.1 Reading and Writing Data in Text Format

pandas features a number of functions for reading tabular data as a DataFrame object. Table 6-1 summarizes some 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
`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

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

I'll give an overview of the mechanics of these functions, which are meant to convert text data into a DataFrame. The optional arguments for these functions may 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.

Because of how messy data in the real world can be, some of the data loading functions (especially `read_csv`) have grown very complex in their options over time. It's normal to feel overwhelmed by the number of different parameters (`read_csv` has over 50 as of this writing). The online pandas documentation has many examples about how each of them works, so if you're struggling to read a particular file, there might be a similar enough example to help you find the right parameters.

Some of these functions, like `pandas.read_csv`, perform *type inference*, because the column data types are not part of the data format. That means you don't necessarily have to specify which columns are numeric, integer, boolean, or string. Other data formats, like HDF5, Feather, and msgpack, have the data types stored in the format.

Handling dates and other custom types can require extra effort. Let's start with a small comma-separated (CSV) text file:

```
In [8]: !cat examples/ex1.csv
a,b,c,d,message
1,2,3,4,hello
```

```
5,6,7,8,world
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.

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

```
In [9]: df = pd.read_csv('examples/ex1.csv')

In [10]: df
Out[10]:
   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 specified the delimiter:

```
In [11]: pd.read_table('examples/ex1.csv', sep=',')
Out[11]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

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

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

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

```
In [13]: pd.read_csv('examples/ex2.csv', header=None)
Out[13]:
   0   1   2   3      4
0  1   2   3   4  hello
1  5   6   7   8  world
2  9  10  11  12    foo

In [14]: pd.read_csv('examples/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
Out[14]:
   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 [15]: names = ['a', 'b', 'c', 'd', 'message']

In [16]: pd.read_csv('examples/ex2.csv', names=names, index_col='message')
Out[16]:
         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, pass a list of column numbers or names:

```
In [17]: !cat examples/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 [18]: parsed = pd.read_csv('examples/csv_mindex.csv',
   ....:                      index_col=['key1', 'key2'])

In [19]: parsed
Out[19]:
           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. Consider a text file that looks like this:

```
In [20]: list(open('examples/ex3.txt'))
Out[20]:
['            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, the fields here are separated by a variable amount of whitespace. In these cases, you can pass a regular expression as a delimiter for `read_table`. This can be expressed by the regular expression `\s+`, so we have then:

```
In [21]: result = pd.read_table('examples/ex3.txt', sep='\s+')

In [22]: result
Out[22]:
            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 a partial listing in Table 6-2). For example, you can skip the first, third, and fourth rows of a file with `skiprows`:

```
In [23]: !cat examples/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 [24]: pd.read_csv('examples/ex4.csv', skiprows=[0, 2, 3])
Out[24]:
   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` and `NULL`:

```
In [25]: !cat examples/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 [26]: result = pd.read_csv('examples/ex5.csv')
```

```
In [27]: result
Out[27]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo

In [28]: pd.isnull(result)
Out[28]:
   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 [29]: result = pd.read_csv('examples/ex5.csv', na_values=['NULL'])

In [30]: result
Out[30]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

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

```
In [31]: sentinels = {'message': ['foo', 'NA'], 'something': ['two']}

In [32]: pd.read_csv('examples/ex5.csv', na_values=sentinels)
Out[32]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       NaN  5   6   NaN   8   world
2     three  9  10  11.0  12     NaN
```

Table 6-2 lists some frequently used options in `pandas.read_csv` and `pandas.read_table`.

Table 6-2. Some 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`

Argument	Description
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(s) 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 (e.g., if date/time split across two columns).
keep_date_col	If joining columns to parse date, keep the joined columns; `False` by default.
converters	Dict containing column number of name mapping to functions (e.g., `{'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); `False` by default.
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 (e.g., `'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.

Before we look at a large file, we make the pandas display settings more compact:

```
In [33]: pd.options.display.max_rows = 10
```

Now we have:

```
In [34]: result = pd.read_csv('examples/ex6.csv')

In [35]: result
Out[35]:
           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
...        ...       ...       ...       ...  ..
9995  2.311896 -0.417070 -1.409599 -0.515821   L
```

```
9996 -0.479893 -0.650419  0.745152 -0.646038   E
9997  0.523331  0.787112  0.486066  1.093156   K
9998 -0.362559  0.598894 -1.843201  0.887292   G
9999 -0.096376 -1.012999 -0.657431 -0.573315   0
[10000 rows x 5 columns]
```

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

```
In [36]: pd.read_csv('examples/ex6.csv', nrows=5)
Out[36]:
        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 a file in pieces, specify a `chunksize` as a number of rows:

```
In [37]: chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

In [38]: chunker
Out[38]: <pandas.io.parsers.TextFileReader at 0x7f6b1e2672e8>
```

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('examples/ex6.csv', chunksize=1000)

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

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

We have then:

```
In [40]: tot[:10]
Out[40]:
E    368.0
X    364.0
L    346.0
O    343.0
Q    340.0
M    338.0
J    337.0
F    335.0
K    334.0
H    330.0
dtype: float64
```

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

Writing Data to Text Format

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

```
In [41]: data = pd.read_csv('examples/ex5.csv')

In [42]: data
Out[42]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

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

```
In [43]: data.to_csv('examples/out.csv')

In [44]: !cat examples/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 prints the text result to the console):

```
In [45]: import sys

In [46]: 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 [47]: 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 [48]: 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 [49]: data.to_csv(sys.stdout, index=False, columns=['a', 'b', 'c'])
a,b,c
1,2,3.0
5,6,
9,10,11.0
```

Series also has a `to_csv` method:

```
In [50]: dates = pd.date_range('1/1/2000', periods=7)

In [51]: ts = pd.Series(np.arange(7), index=dates)

In [52]: ts.to_csv('examples/tseries.csv')

In [53]: !cat examples/tseries.csv
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
```

Working with Delimited Formats

It's possible to load most forms of tabular data 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 [54]: !cat examples/ex7.csv
"a","b","c"
"1","2","3"
"1","2","3"
```

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('examples/ex7.csv')

reader = csv.reader(f)
```

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

```
In [56]: for line in reader:
   ....:     print(line)
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3']
```

From there, it's up to you to do the wrangling necessary to put the data in the form that you need it. Let's take this step by step. First, we read the file into a list of lines:

```
In [57]: with open('examples/ex7.csv') as f:
   ....:     lines = list(csv.reader(f))
```

Then, we split the lines into the header line and the data lines:

```
In [58]: header, values = lines[0], lines[1:]
```

Then we can create a dictionary of data columns using a dictionary comprehension and the expression `zip(*values)`, which transposes rows to columns:

```
In [59]: data_dict = {h: v for h, v in zip(header, zip(*values))}

In [60]: data_dict
Out[60]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

CSV files come in many different flavors. To define a new format with a different delimiter, string quoting convention, or line terminator, we define a simple subclass of `csv.Dialect`:

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

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

We can also give individual CSV dialect parameters 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
`delimiter`	One-character string to separate fields; defaults to `','`.
`lineterminator`	Line terminator for writing; defaults to `'\r\n'`. Reader ignores this and recognizes cross-platform line terminators.
`quotechar`	Quote character for fields with special characters (like a delimiter); default is `'"'`.

Argument	Description
`quoting`	Quoting convention. Options include `csv.QUOTE_ALL` (quote all fields), `csv.QUOTE_MINIMAL` (only fields with special characters like the delimiter), `csv.QUOTE_NONNUMERIC`, and `csv.QUOTE_NONE` (no quoting). See Python's documentation for full details. Defaults to `QUOTE_MINIMAL`.
`skipinitialspace`	Ignore whitespace after each delimiter; default is `False`.
`doublequote`	How to handle quoting character inside a field; if `True`, it is doubled (see online documentation for full detail and behavior).
`escapechar`	String to escape the delimiter if `quoting` is set to `csv.QUOTE_NONE`; 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 free-form 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": 30, "pets": ["Zeus", "Zuko"]},
              {"name": "Katie", "age": 38,
               "pets": ["Sixes", "Stache", "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 [62]: import json

In [63]: result = json.loads(obj)

In [64]: result
Out[64]:
{'name': 'Wes',
 'pet': None,
 'places_lived': ['United States', 'Spain', 'Germany'],
 'siblings': [{'age': 30, 'name': 'Scott', 'pets': ['Zeus', 'Zuko']},
  {'age': 38, 'name': 'Katie', 'pets': ['Sixes', 'Stache', 'Cisco']}]}
```

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

```
In [65]: 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 dicts (which were previously JSON objects) to the DataFrame constructor and select a subset of the data fields:

```
In [66]: siblings = pd.DataFrame(result['siblings'], columns=['name', 'age'])

In [67]: siblings
Out[67]:
    name  age
0  Scott   30
1  Katie   38
```

The `pandas.read_json` can automatically convert JSON datasets in specific arrangements into a Series or DataFrame. For example:

```
In [68]: !cat examples/example.json
[{"a": 1, "b": 2, "c": 3},
 {"a": 4, "b": 5, "c": 6},
 {"a": 7, "b": 8, "c": 9}]
```

The default options for `pandas.read_json` assume that each object in the JSON array is a row in the table:

```
In [69]: data = pd.read_json('examples/example.json')

In [70]: data
Out[70]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

For an extended example of reading and manipulating JSON data (including nested records), see the USDA Food Database example in Chapter 7.

If you need to export data from pandas to JSON, one way is to use the `to_json` methods on Series and DataFrame:

```
In [71]: print(data.to_json())
{"a":{"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":{"0":3,"1":6,"2":9}}

In [72]: print(data.to_json(orient='records'))
[{"a":1,"b":2,"c":3},{"a":4,"b":5,"c":6},{"a":7,"b":8,"c":9}]
```

XML and HTML: Web Scraping

Python has many libraries for reading and writing data in the ubiquitous HTML and XML formats. Examples include lxml, Beautiful Soup, and html5lib. While lxml is comparatively much faster in general, the other libraries can better handle malformed HTML or XML files.

pandas has a built-in function, `read_html`, which uses libraries like lxml and Beautiful Soup to automatically parse tables out of HTML files as DataFrame objects. To show how this works, I downloaded an HTML file (used in the pandas documentation) from the United States FDIC government agency showing bank failures.[1] First, you must install some additional libraries used by `read_html`:

```
conda install lxml
pip install beautifulsoup4 html5lib
```

If you are not using conda, `pip install lxml` will likely also work.

The `pandas.read_html` function has a number of options, but by default it searches for and attempts to parse all tabular data contained within `<table>` tags. The result is a list of DataFrame objects:

```
In [73]: tables = pd.read_html('examples/fdic_failed_bank_list.html')

In [74]: len(tables)
Out[74]: 1

In [75]: failures = tables[0]

In [76]: failures.head()
Out[76]:
                      Bank Name             City  ST   CERT  \
0                   Allied Bank         Mulberry  AR     91
1  The Woodbury Banking Company         Woodbury  GA  11297
2        First CornerStone Bank  King of Prussia  PA  35312
```

1 For the full list, see *https://www.fdic.gov/bank/individual/failed/banklist.html*.

```
3            Trust Company Bank         Memphis  TN   9956
4    North Milwaukee State Bank       Milwaukee  WI  20364
                 Acquiring Institution       Closing Date       Updated Date
0                         Today's Bank  September 23, 2016  November 17, 2016
1                          United Bank     August 19, 2016  November 17, 2016
2  First-Citizens Bank & Trust Company         May 6, 2016  September 6, 2016
3           The Bank of Fayette County      April 29, 2016  September 6, 2016
4  First-Citizens Bank & Trust Company      March 11, 2016      June 16, 2016
```

Because `failures` has many columns, pandas inserts a line break character \.

As you will learn in later chapters, from here we could proceed to do some data cleaning and analysis, like computing the number of bank failures by year:

```
In [77]: close_timestamps = pd.to_datetime(failures['Closing Date'])

In [78]: close_timestamps.dt.year.value_counts()
Out[78]:
2010    157
2009    140
2011     92
2012     51
2008     25
       ...
2004      4
2001      4
2007      3
2003      3
2000      2
Name: Closing Date, Length: 15, dtype: int64
```

Parsing XML with lxml.objectify

XML (eXtensible Markup Language) is another common structured data format supporting hierarchical, nested data with metadata. The book you are currently reading was actually created from a series of large XML documents.

Earlier, I showed the `pandas.read_html` function, which uses either lxml or Beautiful Soup under the hood to parse data from HTML. XML and HTML are structurally similar, but XML is more general. Here, I will show an example of how to use lxml to parse data from a more general XML format.

The New York Metropolitan Transportation Authority (MTA) publishes a number of data series about its bus and train services. 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`:

```python
from lxml import objectify

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

`root.INDICATOR` returns 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):

```python
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 [81]: perf = pd.DataFrame(data)

In [82]: perf.head()
Out[82]:
Empty DataFrame
```

```
Columns: []
Index: []
```

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 io 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 [84]: root
Out[84]: <Element a at 0x7f6b15817748>

In [85]: root.get('href')
Out[85]: 'http://www.google.com'

In [86]: root.text
Out[86]: 'Google'
```

6.2 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 [87]: frame = pd.read_csv('examples/ex1.csv')

In [88]: frame
Out[88]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo

In [89]: 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 [90]: pd.read_pickle('examples/frame_pickle')
Out[90]:
   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.

pandas has built-in support for two more binary data formats: HDF5 and MessagePack. I will give some HDF5 examples in the next section, but I encourage you to explore different file formats to see how fast they are and how well they work for your analysis. Some other storage formats for pandas or NumPy data include:

bcolz
: A compressable column-oriented binary format based on the Blosc compression library.

Feather
: A cross-language column-oriented file format I designed with the R programming community's Hadley Wickham. Feather uses the Apache Arrow columnar memory 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 [92]: frame = pd.DataFrame({'a': np.random.randn(100)})

In [93]: store = pd.HDFStore('mydata.h5')

In [94]: store['obj1'] = frame

In [95]: store['obj1_col'] = frame['a']

In [96]: store
```

```
Out[96]:
<class 'pandas.io.pytables.HDFStore'>
File path: mydata.h5
/obj1                frame        (shape->[100,1])

/obj1_col            series       (shape->[100])

/obj2                frame_table  (typ->appendable,nrows->100,ncols->1,indexers->
[index])
/obj3                frame_table  (typ->appendable,nrows->100,ncols->1,indexers->
[index])
```

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

```
In [97]: store['obj1']
Out[97]:
           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 x 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 [98]: store.put('obj2', frame, format='table')

In [99]: store.select('obj2', where=['index >= 10 and index <= 15'])
Out[99]:
           a
10  1.007189
11 -1.296221
12  0.274992
13  0.228913
14  1.352917
15  0.886429

In [100]: store.close()
```

The `put` is an explicit version of the `store['obj2'] = frame` method but allows us to set other options like the storage format.

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

```
In [101]: frame.to_hdf('mydata.h5', 'obj3', format='table')
```

```
In [102]: pd.read_hdf('mydata.h5', 'obj3', where=['index < 5'])
Out[102]:
          a
0 -0.204708
1  0.478943
2 -0.519439
3 -0.555730
4  1.965781
```

If you are processing data that is stored on remote servers, like Amazon S3 or HDFS, using a different binary format designed for distributed storage like Apache Parquet may be more suitable. Python for Parquet and other such storage formats is still developing, so I do not write about them in this book.

If you work with large quantities of data locally, I would encourage you to explore PyTables and h5py to see how they can suit your needs. Since many data analysis problems are I/O-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 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.

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

```
In [104]: xlsx = pd.ExcelFile('examples/ex1.xlsx')
```

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

```
In [105]: pd.read_excel(xlsx, 'Sheet1')
Out[105]:
   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 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 [106]: frame = pd.read_excel('examples/ex1.xlsx', 'Sheet1')

In [107]: frame
Out[107]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
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 [108]: writer = pd.ExcelWriter('examples/ex2.xlsx')

In [109]: frame.to_excel(writer, 'Sheet1')

In [110]: writer.save()
```

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

```
In [111]: frame.to_excel('examples/ex2.xlsx')
```

6.3 Interacting with 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.

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

```
In [113]: import requests

In [114]: url = 'https://api.github.com/repos/pandas-dev/pandas/issues'

In [115]: resp = requests.get(url)

In [116]: resp
Out[116]: <Response [200]>
```

The Response object's `json` method will return a dictionary containing JSON parsed into native Python objects:

```
In [117]: data = resp.json()

In [118]: data[0]['title']
Out[118]: 'Period does not round down for frequencies less that 1 hour'
```

Each element in `data` is a dictionary containing all of the data found on a GitHub issue page (except for the comments). We can pass `data` directly to DataFrame and extract fields of interest:

```
In [119]: issues = pd.DataFrame(data, columns=['number', 'title',
   .....:                                      'labels', 'state'])

In [120]: issues
Out[120]:
    number                                          title  \
0    17666  Period does not round down for frequencies les...
1    17665            DOC: improve docstring of function where
2    17664              COMPAT: skip 32-bit test on int repr
3    17662                         implement Delegator class
4    17654  BUG: Fix series rename called with str alterin...
..     ...                                               ...
25   17603  BUG: Correctly localize naive datetime strings...
26   17599                    core.dtypes.generic --> cython
27   17596   Merge cdate_range functionality into bdate_range
28   17587  Time Grouper bug fix when applied for list gro...
29   17583  BUG: fix tz-aware DatetimeIndex + TimedeltaInd...
                                             labels state
0                                                [] open
1   [{'id': 134699, 'url': 'https://api.github.com... open
2   [{'id': 563047854, 'url': 'https://api.github.... open
3                                                [] open
4   [{'id': 76811, 'url': 'https://api.github.com/... open
..                                              ...  ...
25  [{'id': 76811, 'url': 'https://api.github.com/... open
26  [{'id': 49094459, 'url': 'https://api.github.c... open
27  [{'id': 35818298, 'url': 'https://api.github.c... open
28  [{'id': 233160, 'url': 'https://api.github.com... open
29  [{'id': 76811, 'url': 'https://api.github.com/... open
[30 rows x 4 columns]
```

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

6.4 Interacting with Databases

In a business setting, most data may not be stored in text or Excel files. SQL-based relational databases (such as SQL Server, PostgreSQL, and MySQL) are in wide use, and many alternative 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 create a SQLite database using Python's built-in `sqlite3` driver:

```
In [121]: import sqlite3

In [122]: query = """
   .....: CREATE TABLE test
```

```
   .....: (a VARCHAR(20), b VARCHAR(20),
   .....:  c REAL,        d INTEGER
   .....: );"""

In [123]: con = sqlite3.connect('mydata.sqlite')

In [124]: con.execute(query)
Out[124]: <sqlite3.Cursor at 0x7f6b12a50f10>

In [125]: con.commit()
```

Then, insert a few rows of data:

```
In [126]: data = [('Atlanta', 'Georgia', 1.25, 6),
   .....:         ('Tallahassee', 'Florida', 2.6, 3),
   .....:         ('Sacramento', 'California', 1.7, 5)]

In [127]: stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

In [128]: con.executemany(stmt, data)
Out[128]: <sqlite3.Cursor at 0x7f6b15c66ce0>

In [129]: con.commit()
```

Most Python SQL drivers (PyODBC, psycopg2, MySQLdb, pymssql, etc.) return a list of tuples when selecting data from a table:

```
In [130]: cursor = con.execute('select * from test')

In [131]: rows = cursor.fetchall()

In [132]: rows
Out[132]:
[('Atlanta', 'Georgia', 1.25, 6),
 ('Tallahassee', 'Florida', 2.6, 3),
 ('Sacramento', '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 [133]: cursor.description
Out[133]:
(('a', None, None, None, None, None, None),
 ('b', None, None, None, None, None, None),
 ('c', None, None, None, None, None, None),
 ('d', None, None, None, None, None, None))

In [134]: pd.DataFrame(rows, columns=[x[0] for x in cursor.description])
Out[134]:
             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. The SQLAlchemy project is a popular Python SQL toolkit that abstracts away many of the common differences between SQL databases. pandas has a `read_sql` function that enables you to read data easily from a general SQLAlchemy connection. Here, we'll connect to the same SQLite database with SQLAlchemy and read data from the table created before:

```
In [135]: import sqlalchemy as sqla

In [136]: db = sqla.create_engine('sqlite:///mydata.sqlite')

In [137]: pd.read_sql('select * from test', db)
Out[137]:
             a           b     c  d
0      Atlanta     Georgia  1.25  6
1  Tallahassee     Florida  2.60  3
2   Sacramento  California  1.70  5
```

6.5 Conclusion

Getting access to data is frequently the first step in the data analysis process. We have looked at a number of useful tools in this chapter that should help you get started. In the upcoming chapters we will dig deeper into data wrangling, data visualization, time series analysis, and other topics.