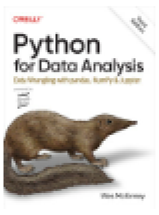

Python for Data Analysis, 3rd Edition

Wes McKinney

Published by O'Reilly Media, Inc.

https://github.com/wesm/pydata-book

Chapter 6. Data Loading, Storage, and File Formats

Reading data and making it accessible (often called *data loading*) is a necessary first step for using most of the tools in this book. The term *parsing* is also sometimes used to describe loading text data and interpreting it as tables and different data types. I'm going to focus 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 fall 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; pandas.read_csv is one of the most frequently used in this book. We will look at binary data formats later in Section 6.2, "Binary Data Formats,".

Function	Description
read_csv	Load delimited data from a file, URL, or file-like object; use comma as default delimiter
read_fwf	Read data in fixed-width column format (i.e., no delimiters)
read_clipboard	Variation of read_csv 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

Function	Description
read_html	Read all tables found in the given HTML document
read_json	Read data from a JSON (JavaScript Object Notation) string representation, file, URL, or file-like object
read_feather	Read the Feather binary file format
read_orc	Read the Apache ORC binary file format
read_parquet	Read the Apache Parquet binary file format
read_pickle	Read an object stored by pandas using the Python pickle format
read_sas	Read a SAS dataset stored in one of the SAS system's custom storage formats
read_spss	Read a data file created by SPSS
read_sql	Read the results of a SQL query (using SQLAlchemy)
read_sql_table	Read a whole SQL table (using SQLAlchemy); equivalent to using a query that selects everything in that table using read_sql
read_stata	Read a dataset from Stata file format
read_xml	Read a table of data from an XML file

Table 6-1. Text and binary data loading functions in pandas

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, arguments you provide, or not at all.

Type inference and data conversion

Includes the user-defined value conversions and custom list of missing value markers.

Date and time parsing

Includes a 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

Includes 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 pandas.read_csv) have accumulated a long list of optional arguments over time. It's normal to feel overwhelmed by the number of different parameters (pandas.read_csv has around 50). The online pandas documentation has many examples about how each of these 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 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, ORC, and Parquet, have the data type information embedded in the format.

Handling dates and other custom types can require extra effort.

Let's start with a small comma-separated values (CSV) text file:

```
In [10]: !cat examples/ex1.csv

a,b,c,d,message

1,2,3,4,hello

5,6,7,8,world

9,10,11,12,foo
```

Note

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 within a Windows terminal (or command line).

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

```
In [11]: df = pd.read_csv("examples/ex1.csv")


In [12]: df

Out[12]:

   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 [13]: !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 [14]: pd.read_csv("examples/ex2.csv", header=None)

Out[14]:

   0   1   2   3      4

0  1   2   3   4  hello

1  5   6   7   8  world

2  9  10  11  12    foo
```

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

Out[15]:

  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 [16]: names = ["a", "b", "c", "d", "message"]


In [17]: pd.read_csv("examples/ex2.csv", names=names, index_col="message")

Out[17]:

       a  b  c  d

message

hello    1  2  3  4

world    5  6  7  8

foo     9  10  11  12
```

If you want to form a hierarchical index (discussed in Section 8.1, “Hierarchical Indexing,”) from multiple columns, pass a list of column numbers or names:

```
In [18]: !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 [19]: parsed = pd.read_csv("examples/csv_mindex.csv",
   ....:                      index_col=["key1", "key2"])

In [20]: parsed
Out[20]:
           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 [21]: !cat examples/ex3.txt
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
```

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 pandas.read_csv. This can be expressed by the regular expression \s+, so we have then:

```
In [22]: result = pd.read_csv("examples/ex3.txt", sep="\s+")


In [23]: result

Out[23]:

          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, pandas.read_csv infers that the first column should be the DataFrame’s index in this special case.

The file parsing 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 [24]: !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 [25]: pd.read_csv("examples/ex4.csv", skiprows=[0, 2, 3])

Out[25]:

   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 reading process. Missing data is usually either not present (empty string) or marked by some *sentinel* (placeholder) value. By default, pandas uses a set of commonly occurring sentinels, such as NA and NULL:

```
In [26]: !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 [27]: result = pd.read_csv("examples/ex5.csv")


In [28]: result

Out[28]:

  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
```

Recall that pandas outputs missing values as NaN, so we have two null or missing values in result:

```
In [29]: pd.isna(result)

Out[29]:

  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 accepts a sequence of strings to add to the default list of strings recognized as missing:

```
In [30]: result = pd.read_csv("examples/ex5.csv", na_values=["NULL"])


In [31]: result

Out[31]:

  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
```

pandas.read_csv has a list of many default NA value representations, but these defaults can be disabled with the keep_default_na option:

```
In [32]: result2 = pd.read_csv("examples/ex5.csv", keep_default_na=False)


In [33]: result2

Out[33]:

  something  a   b   c   d message

0       one  1   2   3   4      NA

1       two  5   6       8   world

2     three  9  10  11  12     foo
```

```
In [34]: result2.isna()

Out[34]:

  something     a     b     c     d  message

0    False  False  False  False  False   False

1    False  False  False  False  False   False

2    False  False  False  False  False   False


In [35]: result3 = pd.read_csv("examples/ex5.csv", keep_default_na=False,

  ....:                    na_values=["NA"])


In [36]: result3

Out[36]:

 something  a   b   c   d message

0     one  1   2   3   4    NaN

1     two  5   6       8   world

2   three  9  10  11  12    foo


In [37]: result3.isna()

Out[37]:

  something     a     b     c     d  message

0    False  False  False  False  False    True

1    False  False  False  False  False   False

2    False  False  False  False  False   False
```

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

```
In [38]: sentinels = {"message": ["foo", "NA"], "something": ["two"]}


In [39]: pd.read_csv("examples/ex5.csv", na_values=sentinels,
  ....:         keep_default_na=False)
Out[39]:
 something  a   b   c   d message
0      one  1   2   3   4     NaN
1      NaN  5   6       8   world
2    three  9  10  11  12     NaN
```

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

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.
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. They are added to the default list unless keep_default_na=False is passed.

Argument	Description
keep_default_na	Whether to use the default NA value list or not (True by default).
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 names 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	Dictionary containing column number or 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 (not counting the header).
iterator	Return a TextFileReader object for reading the file piecemeal. This object can also be used with the with statement.
chunksize	For iteration, size of file chunks.
skip_footer	Number of lines to ignore at end of file.

Argument	Description
verbose	Print various parsing information, like the time spent in each stage of the file conversion and memory use information.
encoding	Text encoding (e.g., "utf-8 for UTF-8 encoded text). Defaults to "utf-8" if None.
squeeze	If the parsed data contains only one column, return a Series.
thousands	Separator for thousands (e.g., "," or "."); default is None.
decimal	Decimal separator in numbers (e.g., "." or ","); default is ".".
engine	CSV parsing and conversion engine to use; can be one of "c", "python", or "pyarrow". The default is "c", though the newer "pyarrow" engine can parse some files much faster. The "python" engine is slower but supports some features that the other engines do not.

Table 6-2. Some pandas.read_csv function arguments

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 want to read only 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 [40]: pd.options.display.max_rows = 10
```

Now we have:

```
In [41]: result = pd.read_csv("examples/ex6.csv")
In [42]: result
Out[42]:
         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]
```

The elipsis marks ... indicate that rows in the middle of the DataFrame have been omitted.

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

```
In [43]: pd.read_csv("examples/ex6.csv", nrows=5)
Out[43]:
       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 [44]: chunker = pd.read_csv("examples/ex6.csv", chunksize=1000)


In [45]: type(chunker)

Out[45]: pandas.io.parsers.readers.TextFileReader
```

The TextFileReader object returned by pandas.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([], dtype='int64')

for piece in chunker:

   tot = tot.add(piece["key"].value_counts(), fill_value=0)


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

We have then:

```
In [47]: tot[:10]

Out[47]:

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
```

TextFileReader 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 [48]: data = pd.read_csv("examples/ex5.csv")

In [49]: data
Out[49]:
  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 [50]: data.to_csv("examples/out.csv")

In [51]: !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 rather than a file):

```
In [52]: import sys


In [53]: 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 [54]: 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 [55]: 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 [56]: data.to_csv(sys.stdout, index=False, columns=["a", "b", "c"])

a,b,c

1,2,3.0

5,6,

9,10,11.0
```

Working with Other Delimited Formats

It’s possible to load most forms of tabular data from disk using functions like pandas.read_csv. 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 pandas.read_csv. To illustrate the basic tools, consider a small CSV file:

```
In [57]: !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:

```
In [58]: import csv


In [59]: f = open("examples/ex7.csv")


In [60]: reader = csv.reader(f)
```

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

```
In [61]: for line in reader:
   ....:     print(line)
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3']

In [62]: f.close()
```

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

```
In [63]: 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 [64]: header, values = lines[0], lines[1:]
```

Then we can create a dictionary of data columns using a dictionary comprehension and the expression zip(*values) (beware that this will use a lot of memory on large files), which transposes rows to columns:

```
In [65]: data_dict = {h: v for h, v in zip(header, zip(*values))}

In [66]: data_dict
Out[66]: {'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 could 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 could 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.

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.

Argument	Description
quotechar	Quote character for fields with special characters (like a delimiter); default is '"'.
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.

Table 6-3. CSV dialect options

Note

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 the string's split method or the regular expression method re.split. Thankfully, pandas.read_csv is capable of doing almost anything you need if you pass the necessary options, so you only rarely will have to parse files by hand.

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",

 "cities_lived": ["Akron", "Nashville", "New York", "San Francisco"],

 "pet": null,

 "siblings": [{"name": "Scott", "age": 34, "hobbies": ["guitars", "soccer"]},

             {"name": "Katie", "age": 42, "hobbies": ["diving", "art"]}]

}

"""
```

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 (dictionaries), 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 [68]: import json


In [69]: result = json.loads(obj)


In [70]: result

Out[70]:

{'name': 'Wes',

 'cities_lived': ['Akron', 'Nashville', 'New York', 'San Francisco'],

 'pet': None,

 'siblings': [{'name': 'Scott',

   'age': 34,
  'hobbies': ['guitars', 'soccer']},
 {'name': 'Katie', 'age': 42, 'hobbies': ['diving', 'art']}]}
```

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

```
In [71]: asjson = json.dumps(result)

In [72]: asjson
Out[72]: '{"name": "Wes", "cities_lived": ["Akron", "Nashville", "New York", "San
 Francisco"], "pet": null, "siblings": [{"name": "Scott", "age": 34, "hobbies": [
"guitars", "soccer"]}, {"name": "Katie", "age": 42, "hobbies": ["diving", "art"]}
]}'
```

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 dictionaries (which were previously JSON objects) to the DataFrame constructor and select a subset of the data fields:

```
In [73]: siblings = pd.DataFrame(result["siblings"], columns=["name", "age"])

In [74]: siblings
Out[74]:
   name  age
0  Scott   34
1  Katie   42
```

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

```
In [75]: !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 [76]: data = pd.read_json("examples/example.json")


In [77]: data
Out[77]:
   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 13.

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

```
In [78]: data.to_json(sys.stdout)
{"a":{"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":{"0":3,"1":6,"2":9}}
In [79]: data.to_json(sys.stdout, 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, pandas.read_html, which uses all of these libraries 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 US FDIC showing bank failures.[1] First, you must install some additional libraries used by read_html:

```
conda install lxml beautifulsoup4 html5lib
```

If you are not using conda, pip install lxml should 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 [80]: tables = pd.read_html("examples/fdic_failed_bank_list.html")


In [81]: len(tables)

Out[81]: 1


In [82]: failures = tables[0]


In [83]: failures.head()

Out[83]:

                   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

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 [84]: close_timestamps = pd.to_datetime(failures["Closing Date"])

In [85]: close_timestamps.dt.year.value_counts()
Out[85]:
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 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.

For many years, the New York Metropolitan Transportation Authority (MTA) published a number of data series about its bus and train services in XML format. 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:

```
In [86]: from lxml import objectify

In [87]: path = "datasets/mta_perf/Performance_MNR.xml"

In [88]: with open(path) as f:
   ....:     parsed = objectify.parse(f)

In [89]: root = parsed.getroot()
```

root.INDICATOR returns a generator yielding each <INDICATOR> XML element. For each record, we can populate a dictionary of tag names (like YTD_ACTUAL) to data values (excluding a few tags) by running the following code:

```
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 dictionaries into a DataFrame:

```
In [91]: perf = pd.DataFrame(data)


In [92]: perf.head()

Out[92]:

        AGENCY_NAME                 INDICATOR_NAME  \

0  Metro-North Railroad  On-Time Performance (West of Hudson)

1  Metro-North Railroad  On-Time Performance (West of Hudson)

2  Metro-North Railroad  On-Time Performance (West of Hudson)

3  Metro-North Railroad  On-Time Performance (West of Hudson)

4  Metro-North Railroad  On-Time Performance (West of Hudson)

                                            DESCRIPTION  \

0  Percent of commuter trains that arrive at their destinations within 5 m...

1  Percent of commuter trains that arrive at their destinations within 5 m...

2  Percent of commuter trains that arrive at their destinations within 5 m...

3  Percent of commuter trains that arrive at their destinations within 5 m...

4  Percent of commuter trains that arrive at their destinations within 5 m...

  PERIOD_YEAR  PERIOD_MONTH         CATEGORY FREQUENCY INDICATOR_UNIT  \

0       2008            1  Service Indicators      M          %

1       2008            2  Service Indicators      M          %

2       2008            3  Service Indicators      M          %

3       2008            4  Service Indicators      M          %

4       2008            5  Service Indicators      M          %

  YTD_TARGET YTD_ACTUAL MONTHLY_TARGET MONTHLY_ACTUAL

0     95.0      96.9         95.0         96.9

1     95.0      96.0         95.0         95.0

2     95.0      96.3         95.0         96.9

3     95.0      96.8         95.0         98.3

4     95.0      96.6         95.0         95.8
```

pandas’s pandas.read_xml function turns this process into a one-line expression:

```
In [93]: perf2 = pd.read_xml(path)


In [94]: perf2.head()

Out[94]:

  INDICATOR_SEQ  PARENT_SEQ        AGENCY_NAME  \

0         28445         NaN  Metro-North Railroad

1         28445         NaN  Metro-North Railroad

2         28445         NaN  Metro-North Railroad

3         28445         NaN  Metro-North Railroad

4         28445         NaN  Metro-North Railroad

                  INDICATOR_NAME  \

0  On-Time Performance (West of Hudson)

1  On-Time Performance (West of Hudson)

2  On-Time Performance (West of Hudson)

3  On-Time Performance (West of Hudson)

4  On-Time Performance (West of Hudson)

                                         DESCRIPTION  \

0  Percent of commuter trains that arrive at their destinations within 5 m...

1  Percent of commuter trains that arrive at their destinations within 5 m...

2  Percent of commuter trains that arrive at their destinations within 5 m...

3  Percent of commuter trains that arrive at their destinations within 5 m...

4  Percent of commuter trains that arrive at their destinations within 5 m...

  PERIOD_YEAR  PERIOD_MONTH          CATEGORY FREQUENCY DESIRED_CHANGE  \

0        2008             1  Service Indicators         M              U

1        2008             2  Service Indicators         M              U

2        2008             3  Service Indicators         M              U

3        2008             4  Service Indicators         M              U

4        2008             5  Service Indicators         M              U
```

```
 INDICATOR_UNIT  DECIMAL_PLACES YTD_TARGET YTD_ACTUAL MONTHLY_TARGET  \
0          %          1     95.00     96.90        95.00
1          %          1     95.00     96.00        95.00
2          %          1     95.00     96.30        95.00
3          %          1     95.00     96.80        95.00
4          %          1     95.00     96.60        95.00
 MONTHLY_ACTUAL
0       96.90
1       95.00
2       96.90
3       98.30
4       95.80
```

For more complex XML documents, refer to the docstring for pandas.read_xml which describes how to do selections and filters to extract a particular table of interest.

6.2 Binary Data Formats

One simple way to store (or *serialize*) data in binary format is using Python's built-in pickle module. pandas objects all have a to_pickle method that writes the data to disk in pickle format:

```
In [95]: frame = pd.read_csv("examples/ex1.csv")


In [96]: frame

Out[96]:

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


In [97]: frame.to_pickle("examples/frame_pickle")
```

Pickle files are in general readable only in Python. 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 [98]: pd.read_pickle("examples/frame_pickle")
```

```
Out[98]:

  a  b  c  d message

0 1  2  3  4  hello

1 5  6  7  8  world

2 9  10  11  12    foo
```

Caution

pickle is recommended only 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. pandas has 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 several other open source binary data formats, such as HDF5, ORC, and Apache Parquet. For example, if you install the pyarrow package (conda install pyarrow), then you can read Parquet files with pandas.read_parquet:

```
In [100]: fec = pd.read_parquet('datasets/fec/fec.parquet')
```

I will give some HDF5 examples in "Using HDF5 Format". I encourage you to explore different file formats to see how fast they are and how well they work for your analysis.

Reading Microsoft Excel Files

pandas also supports reading tabular data stored in Excel 2003 (and higher) files using either the pandas.ExcelFile class or pandas.read_excel function. Internally, these tools use the add-on packages xlrd and openpyxl to read old-style XLS and newer XLSX files, respectively. These must be installed separately from pandas using pip or conda:

```
conda install openpyxl xlrd
```

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

```
In [101]: xlsx = pd.ExcelFile("examples/ex1.xlsx")
```

This object can show you the list of available sheet names in the file:

```
In [102]: xlsx.sheet_names

Out[102]: ['Sheet1']
```

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

```
In [103]: xlsx.parse(sheet_name="Sheet1")

Out[103]:

  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
```

This Excel table has an index column, so we can indicate that with
the index_col argument:

```
In [104]: xlsx.parse(sheet_name="Sheet1", index_col=0)

Out[104]:

  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 pandas.ExcelFile, but you can also simply pass the filename to pandas.read_excel:

```
In [105]: frame = pd.read_excel("examples/ex1.xlsx", sheet_name="Sheet1")


In [106]: frame

Out[106]:

   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 the pandas object's to_excel method:

```
In [107]: writer = pd.ExcelWriter("examples/ex2.xlsx")


In [108]: frame.to_excel(writer, "Sheet1")


In [109]: writer.save()
```

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

```
In [110]: frame.to_excel("examples/ex2.xlsx")
```

Using HDF5 Format

HDF5 is a respected 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 datasets that don’t fit into memory, as you can efficiently read and write small sections of much larger arrays.

To get started with HDF5 and pandas, you must first install PyTables by installing the tables package with conda:

```
conda install pytables
```

Note

Note that the PyTables package is called “tables” in PyPI, so if you install with pip you will have to run pip install tables.

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 objects. The HDFStore class works like a dictionary and handles the low-level details:

```
In [113]: frame = pd.DataFrame({"a": np.random.standard_normal(100)})


In [114]: store = pd.HDFStore("examples/mydata.h5")


In [115]: store["obj1"] = frame


In [116]: store["obj1_col"] = frame["a"]


In [117]: store

Out[117]:

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

File path: examples/mydata.h5
```

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

```
In [118]: store["obj1"]

Out[118]:

          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 default is "fixed"). The latter is generally slower, but it supports query operations using a special syntax:

```
In [119]: store.put("obj2", frame, format="table")


In [120]: store.select("obj2", where=["index >= 10 and index <= 15"])

Out[120]:

          a

10  1.007189

11 -1.296221

12  0.274992

13  0.228913
```

```
14  1.352917
15  0.886429

In [121]: 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 [122]: frame.to_hdf("examples/mydata.h5", "obj3", format="table")

In [123]: pd.read_hdf("examples/mydata.h5", "obj3", where=["index < 5"])
Out[123]:
        a
0 -0.204708
1  0.478943
2 -0.519439
3 -0.555730
4  1.965781
```

If you'd like, you can delete the HDF5 file you created, like so:

```
In [124]: import os

In [125]: os.remove("examples/mydata.h5")
```

Note

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.

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.

Caution

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.

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 method that I recommend is the requests package, which can be installed with pip or conda:

```
conda install requests
```

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 [126]: import requests


In [127]: url = "https://api.github.com/repos/pandas-dev/pandas/issues"


In [128]: resp = requests.get(url)


In [129]: resp.raise_for_status()


In [130]: resp
Out[130]: <Response [200]>
```

It’s a good practice to always call raise_for_status after using requests.get to check for HTTP errors.

The response object’s json method will return a Python object containing the parsed JSON data as a dictionary or list (depending on what JSON is returned):

```
In [131]: data = resp.json()


In [132]: data[0]["title"]

Out[132]: 'ENH: add `regex=False` option to pandas.Series.str.match and fullmatch

 (like in str.contains)'
```

Since the results retrieved are based on real-time data, what you see when you run this code will almost definitely be different.

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 pandas.DataFrame and extract fields of interest:

```
In [133]: issues = pd.DataFrame(data, columns=["number", "title",

   .....:                              "labels", "state"])


In [134]: issues

Out[134]:

   number  \

0   48086

1   48085

2   48084

3   48083

4   48082

..    ...

25   48052

26   48051

27   48050

28   48048
```

```
29  48046

                                               title  \
0  ENH: add `regex=False` option to pandas.Series.str.match and fullmatch ...
1                 ENH: Add axis parameter to add_prefix and add_suffix
2  ENH: alleviate issues with inconsistent multi-level joining on a 0-colu...
3  BUG: pd.cut creates NaNs instead of expected bins for Timestamps. Behav...
4                           REF: avoid internals in merge code
..                                                ...
25                    BUG: Weird Behavior of astype(bool) for np.nan
26                   REF: dont alter self in _validate_specification
27    ENH: allow user to infer SAS file encoding; add correct encoding names
28  ENH: Add option to `read_sas` to infer encoding from file, then use enc...
29                         CI: setuptools 64.0.0 breaks our builds

                                              labels  \
0  [{'id': 76812, 'node_id': 'MDU6TGFiZWw3NjgxMg==', 'url': 'https://api.g...
1                                                 []
2  [{'id': 76812, 'node_id': 'MDU6TGFiZWw3NjgxMg==', 'url': 'https://api.g...
3  [{'id': 76811, 'node_id': 'MDU6TGFiZWw3NjgxMQ==', 'url': 'https://api.g...
4                                                 []
..                                                ...
25  [{'id': 2822342, 'node_id': 'MDU6TGFiZWwyODIyMzQy', 'url': 'https://api...
26  [{'id': 127681, 'node_id': 'MDU6TGFiZWwxMjc2ODE=', 'url': 'https://api....
27  [{'id': 258745163, 'node_id': 'MDU6TGFiZWwyNTg3NDUxNjM=', 'url': 'https...
28  [{'id': 76812, 'node_id': 'MDU6TGFiZWw3NjgxMg==', 'url': 'https://api.g...
29  [{'id': 48070600, 'node_id': 'MDU6TGFiZWw0ODA3MDYwMA==', 'url': 'https:...

  state
0  open
1  open
2  open
3  open
4  open
```

```
..  ...

25  open

26  open

27  open

28  open

29  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 more convenient analysis.

6.4 Interacting with Databases

In a business setting, a lot of 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.

pandas has some functions to simplify loading the results of a SQL query into a DataFrame. As an example, I'll create a SQLite3 database using Python's built-in sqlite3 driver:

```
In [135]: import sqlite3


In [136]: query = """

   .....: CREATE TABLE test

   .....: (a VARCHAR(20), b VARCHAR(20),

   .....:  c REAL,       d INTEGER

   .....: );"""


In [137]: con = sqlite3.connect("mydata.sqlite")


In [138]: con.execute(query)

Out[138]: <sqlite3.Cursor at 0x7f314395a5c0>
```

```
In [139]: con.commit()
```

Then, insert a few rows of data:

```
In [140]: data = [("Atlanta", "Georgia", 1.25, 6),
  .....:         ("Tallahassee", "Florida", 2.6, 3),
  .....:         ("Sacramento", "California", 1.7, 5)]

In [141]: stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

In [142]: con.executemany(stmt, data)
Out[142]: <sqlite3.Cursor at 0x7f3143940240>

In [143]: con.commit()
```

Most Python SQL drivers return a list of tuples when selecting data from a table:

```
In [144]: cursor = con.execute("SELECT * FROM test")

In [145]: rows = cursor.fetchall()

In [146]: rows
Out[146]:
[('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. Note that for SQLite3, the cursor description only provides column names (the other fields, which are part of Python's Database API specification, are None), but for some other database drivers, more column information is provided:

```
In [147]: cursor.description

Out[147]:

(('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 [148]: pd.DataFrame(rows, columns=[x[0] for x in cursor.description])

Out[148]:

            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. You can install SQLAlchemy with conda like so:

```
conda install sqlalchemy
```

Now, we'll connect to the same SQLite database with SQLAlchemy and read data from the table created before:

```
In [149]: import sqlalchemy as sqla
In [150]: db = sqla.create_engine("sqlite:///mydata.sqlite")

In [151]: pd.read_sql("SELECT * FROM test", db)
Out[151]:
             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.