CHAPTER 6

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
`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 (that is, no delimiters)
`read_clipboard`	Version of `read_table` 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, drop the joined columns. Default True
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
`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 `'"'`.
`quoting`	Quoting convention. Options include `csv.QUOTE_ALL` (quote all fields), `csv.QUOTE_MINIMAL` (only fields with special characters like the delimiter),

Argument	Description
	`csv.QUOTE_NONNUMERIC`, and `csv.QUOTE_NON` (no quoting). See Python's documentation for full details. Defaults to `QUOTE_MINIMAL`.
`skipinitialspace`	Ignore whitespace after each delimiter. Default `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 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]:
Empty DataFrame
Columns: array([], dtype=int64)
Index: array([], dtype=int64)
```

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'
```

```
 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, psycopg2, 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, None),
 ('b', None, None, None, None, None, None),
 ('c', None, None, None, None, None, None),
 ('d', None, 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)
```

CHAPTER 7

Data Wrangling: Clean, Transform, Merge, Reshape

Much of the programming work in data analysis and modeling is spent on data preparation: loading, cleaning, transforming, and rearranging. Sometimes the way that data is stored in files or databases is not the way you need it for a data processing application. Many people choose to do ad hoc processing of data from one form to another using a general purpose programming, like Python, Perl, R, or Java, or UNIX text processing tools like sed or awk. Fortunately, pandas along with the Python standard library provide you with a high-level, flexible, and high-performance set of core manipulations and algorithms to enable you to wrangle data into the right form without much trouble.

If you identify a type of data manipulation that isn't anywhere in this book or elsewhere in the pandas library, feel free to suggest it on the mailing list or GitHub site. Indeed, much of the design and implementation of pandas has been driven by the needs of real world applications.

Combining and Merging Data Sets

Data contained in pandas objects can be combined together in a number of built-in ways:

- `pandas.merge` connects rows in DataFrames based on one or more keys. This will be familiar to users of SQL or other relational databases, as it implements database *join* operations.
- `pandas.concat` glues or stacks together objects along an axis.
- `combine_first` instance method enables splicing together overlapping data to fill in missing values in one object with values from another.

I will address each of these and give a number of examples. They'll be utilized in examples throughout the rest of the book.

Database-style DataFrame Merges

Merge or *join* operations combine data sets by linking rows using one or more *keys*. These operations are central to relational databases. The `merge` function in pandas is the main entry point for using these algorithms on your data.

Let's start with a simple example:

```
In [15]: df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                  'data1': range(7)})

In [16]: df2 = DataFrame({'key': ['a', 'b', 'd'],
   ....:                  'data2': range(3)})

In [17]: df1           In [18]: df2
Out[17]:               Out[18]:
   data1 key              data2 key
0      0   b           0      0   a
1      1   b           1      1   b
2      2   a           2      2   d
3      3   c
4      4   a
5      5   a
6      6   b
```

This is an example of a *many-to-one* merge situation; the data in `df1` has multiple rows labeled `a` and `b`, whereas `df2` has only one row for each value in the `key` column. Calling `merge` with these objects we obtain:

```
In [19]: pd.merge(df1, df2)
Out[19]:
   data1 key  data2
0      2   a      0
1      4   a      0
2      5   a      0
3      0   b      1
4      1   b      1
5      6   b      1
```

Note that I didn't specify which column to join on. If not specified, `merge` uses the overlapping column names as the keys. It's a good practice to specify explicitly, though:

```
In [20]: pd.merge(df1, df2, on='key')
Out[20]:
   data1 key  data2
0      2   a      0
1      4   a      0
2      5   a      0
3      0   b      1
4      1   b      1
5      6   b      1
```

If the column names are different in each object, you can specify them separately:

```
In [21]: df3 = DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                  'data1': range(7)})
```

```
In [22]: df4 = DataFrame({'rkey': ['a', 'b', 'd'],
   ....:                  'data2': range(3)})

In [23]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')
Out[23]:
   data1 lkey  data2 rkey
0      2    a      0    a
1      4    a      0    a
2      5    a      0    a
3      0    b      1    b
4      1    b      1    b
5      6    b      1    b
```

You probably noticed that the `'c'` and `'d'` values and associated data are missing from the result. By default `merge` does an `'inner'` join; the keys in the result are the intersection. Other possible options are `'left'`, `'right'`, and `'outer'`. The outer join takes the union of the keys, combining the effect of applying both left and right joins:

```
In [24]: pd.merge(df1, df2, how='outer')
Out[24]:
   data1 key  data2
0      2   a      0
1      4   a      0
2      5   a      0
3      0   b      1
4      1   b      1
5      6   b      1
6      3   c    NaN
7    NaN   d      2
```

Many-to-many merges have well-defined though not necessarily intuitive behavior. Here's an example:

```
In [25]: df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   ....:                  'data1': range(6)})

In [26]: df2 = DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
   ....:                  'data2': range(5)})

In [27]: df1          In [28]: df2
Out[27]:              Out[28]:
   data1 key             data2 key
0      0   b          0      0   a
1      1   b          1      1   b
2      2   a          2      2   a
3      3   c          3      3   b
4      4   a          4      4   d
5      5   b

In [29]: pd.merge(df1, df2, on='key', how='left')
Out[29]:
    data1 key  data2
0       2   a      0
1       2   a      2
```

```
2      4  a    0
3      4  a    2
4      0  b    1
5      0  b    3
6      1  b    1
7      1  b    3
8      5  b    1
9      5  b    3
10     3  c  NaN
```

Many-to-many joins form the Cartesian product of the rows. Since there were 3 `'b'` rows in the left DataFrame and 2 in the right one, there are 6 `'b'` rows in the result. The join method only affects the distinct key values appearing in the result:

```
In [30]: pd.merge(df1, df2, how='inner')
Out[30]:
   data1 key  data2
0      2   a      0
1      2   a      2
2      4   a      0
3      4   a      2
4      0   b      1
5      0   b      3
6      1   b      1
7      1   b      3
8      5   b      1
9      5   b      3
```

To merge with multiple keys, pass a list of column names:

```
In [31]: left = DataFrame({'key1': ['foo', 'foo', 'bar'],
   ....:                   'key2': ['one', 'two', 'one'],
   ....:                   'lval': [1, 2, 3]})

In [32]: right = DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
   ....:                    'key2': ['one', 'one', 'one', 'two'],
   ....:                    'rval': [4, 5, 6, 7]})

In [33]: pd.merge(left, right, on=['key1', 'key2'], how='outer')
Out[33]:
  key1 key2  lval  rval
0  bar  one     3     6
1  bar  two   NaN     7
2  foo  one     1     4
3  foo  one     1     5
4  foo  two     2   NaN
```

To determine which key combinations will appear in the result depending on the choice of merge method, think of the multiple keys as forming an array of tuples to be used as a single join key (even though it's not actually implemented that way).

When joining columns-on-columns, the indexes on the passed DataFrame objects are discarded.

A last issue to consider in merge operations is the treatment of overlapping column names. While you can address the overlap manually (see the later section on renaming axis labels), `merge` has a `suffixes` option for specifying strings to append to overlapping names in the left and right DataFrame objects:

```
In [34]: pd.merge(left, right, on='key1')
Out[34]:
  key1 key2_x  lval key2_y  rval
0  bar    one     3    one     6
1  bar    one     3    two     7
2  foo    one     1    one     4
3  foo    one     1    one     5
4  foo    two     2    one     4
5  foo    two     2    one     5

In [35]: pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
Out[35]:
  key1 key2_left  lval key2_right  rval
0  bar       one     3        one     6
1  bar       one     3        two     7
2  foo       one     1        one     4
3  foo       one     1        one     5
4  foo       two     2        one     4
5  foo       two     2        one     5
```

See Table 7-1 for an argument reference on `merge`. Joining on index is the subject of the next section.

Table 7-1. merge function arguments

Argument	Description
`left`	DataFrame to be merged on the left side
`right`	DataFrame to be merged on the right side
`how`	One of `'inner'`, `'outer'`, `'left'` or `'right'`. `'inner'` by default
`on`	Column names to join on. Must be found in both DataFrame objects. If not specified and no other join keys given, will use the intersection of the column names in `left` and `right` as the join keys
`left_on`	Columns in `left` DataFrame to use as join keys
`right_on`	Analogous to `left_on` for `left` DataFrame
`left_index`	Use row index in `left` as its join key (or keys, if a MultiIndex)
`right_index`	Analogous to `left_index`
`sort`	Sort merged data lexicographically by join keys; `True` by default. Disable to get better performance in some cases on large datasets
`suffixes`	Tuple of string values to append to column names in case of overlap; defaults to `('_x', '_y')`. For example, if `'data'` in both DataFrame objects, would appear as `'data_x'` and `'data_y'` in result
`copy`	If `False`, avoid copying data into resulting data structure in some exceptional cases. By default always copies

Merging on Index

In some cases, the merge key or keys in a DataFrame will be found in its index. In this case, you can pass `left_index=True` or `right_index=True` (or both) to indicate that the index should be used as the merge key:

```
In [36]: left1 = DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],
   ....:                    'value': range(6)})

In [37]: right1 = DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])

In [38]: left1           In [39]: right1
Out[38]:                 Out[39]:
  key  value                group_val
0   a      0             a        3.5
1   b      1             b        7.0
2   a      2
3   a      3
4   b      4
5   c      5

In [40]: pd.merge(left1, right1, left_on='key', right_index=True)
Out[40]:
  key  value  group_val
0   a      0        3.5
2   a      2        3.5
3   a      3        3.5
1   b      1        7.0
4   b      4        7.0
```

Since the default merge method is to intersect the join keys, you can instead form the union of them with an outer join:

```
In [41]: pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
Out[41]:
  key  value  group_val
0   a      0        3.5
2   a      2        3.5
3   a      3        3.5
1   b      1        7.0
4   b      4        7.0
5   c      5        NaN
```

With hierarchically-indexed data, things are a bit more complicated:

```
In [42]: lefth = DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
   ....:                     'key2': [2000, 2001, 2002, 2001, 2002],
   ....:                     'data': np.arange(5.)})

In [43]: righth = DataFrame(np.arange(12).reshape((6, 2)),
   ....:                    index=[['Nevada', 'Nevada', 'Ohio', 'Ohio', 'Ohio', 'Ohio'],
   ....:                           [2001, 2000, 2000, 2000, 2001, 2002]],
   ....:                    columns=['event1', 'event2'])

In [44]: lefth                  In [45]: righth
Out[44]:                        Out[45]:
```

```
   data    key1  key2                    event1  event2
0     0    Ohio  2000        Nevada 2001      0       1
1     1    Ohio  2001               2000      2       3
2     2    Ohio  2002        Ohio   2000      4       5
3     3  Nevada  2001               2000      6       7
4     4  Nevada  2002               2001      8       9
                                    2002     10      11
```

In this case, you have to indicate multiple columns to merge on as a list (pay attention to the handling of duplicate index values):

```
In [46]: pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True)
Out[46]:
   data    key1  key2  event1  event2
3     3  Nevada  2001       0       1
0     0    Ohio  2000       4       5
0     0    Ohio  2000       6       7
1     1    Ohio  2001       8       9
2     2    Ohio  2002      10      11

In [47]: pd.merge(lefth, righth, left_on=['key1', 'key2'],
   ....:          right_index=True, how='outer')
Out[47]:
   data    key1  key2  event1  event2
4   NaN  Nevada  2000       2       3
3     3  Nevada  2001       0       1
4     4  Nevada  2002     NaN     NaN
0     0    Ohio  2000       4       5
0     0    Ohio  2000       6       7
1     1    Ohio  2001       8       9
2     2    Ohio  2002      10      11
```

Using the indexes of both sides of the merge is also not an issue:

```
In [48]: left2 = DataFrame([[1., 2.], [3., 4.], [5., 6.]], index=['a', 'c', 'e'],
   ....:                   columns=['Ohio', 'Nevada'])

In [49]: right2 = DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
   ....:                    index=['b', 'c', 'd', 'e'], columns=['Missouri', 'Alabama'])

In [50]: left2          In [51]: right2
Out[50]:                Out[51]:
   Ohio  Nevada            Missouri  Alabama
a     1       2         b         7        8
c     3       4         c         9       10
e     5       6         d        11       12
                        e        13       14

In [52]: pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
Out[52]:
   Ohio  Nevada  Missouri  Alabama
a     1       2       NaN      NaN
b   NaN     NaN         7        8
c     3       4         9       10
d   NaN     NaN        11       12
e     5       6        13       14
```

DataFrame has a more convenient `join` instance for merging by index. It can also be used to combine together many DataFrame objects having the same or similar indexes but non-overlapping columns. In the prior example, we could have written:

```
In [53]: left2.join(right2, how='outer')
Out[53]:
   Ohio  Nevada  Missouri  Alabama
a     1       2       NaN      NaN
b   NaN     NaN         7        8
c     3       4         9       10
d   NaN     NaN        11       12
e     5       6        13       14
```

In part for legacy reasons (much earlier versions of pandas), DataFrame's `join` method performs a left join on the join keys. It also supports joining the index of the passed DataFrame on one of the columns of the calling DataFrame:

```
In [54]: left1.join(right1, on='key')
Out[54]:
  key  value  group_val
0   a      0        3.5
1   b      1        7.0
2   a      2        3.5
3   a      3        3.5
4   b      4        7.0
5   c      5        NaN
```

Lastly, for simple index-on-index merges, you can pass a list of DataFrames to `join` as an alternative to using the more general `concat` function described below:

```
In [55]: another = DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
   ....:                     index=['a', 'c', 'e', 'f'], columns=['New York', 'Oregon'])

In [56]: left2.join([right2, another])
Out[56]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a     1       2       NaN      NaN         7       8
c     3       4         9       10         9      10
e     5       6        13       14        11      12

In [57]: left2.join([right2, another], how='outer')
Out[57]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a     1       2       NaN      NaN         7       8
b   NaN     NaN         7        8       NaN     NaN
c     3       4         9       10         9      10
d   NaN     NaN        11       12       NaN     NaN
e     5       6        13       14        11      12
f   NaN     NaN       NaN      NaN        16      17
```

Concatenating Along an Axis

Another kind of data combination operation is alternatively referred to as concatenation, binding, or stacking. NumPy has a `concatenate` function for doing this with raw NumPy arrays:

```
In [58]: arr = np.arange(12).reshape((3, 4))

In [59]: arr
Out[59]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [60]: np.concatenate([arr, arr], axis=1)
Out[60]:
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

In the context of pandas objects such as Series and DataFrame, having labeled axes enable you to further generalize array concatenation. In particular, you have a number of additional things to think about:

- If the objects are indexed differently on the other axes, should the collection of axes be unioned or intersected?
- Do the groups need to be identifiable in the resulting object?
- Does the concatenation axis matter at all?

The `concat` function in pandas provides a consistent way to address each of these concerns. I'll give a number of examples to illustrate how it works. Suppose we have three Series with no index overlap:

```
In [61]: s1 = Series([0, 1], index=['a', 'b'])

In [62]: s2 = Series([2, 3, 4], index=['c', 'd', 'e'])

In [63]: s3 = Series([5, 6], index=['f', 'g'])
```

Calling `concat` with these object in a list glues together the values and indexes:

```
In [64]: pd.concat([s1, s2, s3])
Out[64]:
a    0
b    1
c    2
d    3
e    4
f    5
g    6
```

By default concat works along axis=0, producing another Series. If you pass axis=1, the result will instead be a DataFrame (axis=1 is the columns):

```
In [65]: pd.concat([s1, s2, s3], axis=1)
Out[65]:
     0    1    2
a    0  NaN  NaN
b    1  NaN  NaN
c  NaN    2  NaN
d  NaN    3  NaN
e  NaN    4  NaN
f  NaN  NaN    5
g  NaN  NaN    6
```

In this case there is no overlap on the other axis, which as you can see is the sorted union (the 'outer' join) of the indexes. You can instead intersect them by passing join='inner':

```
In [66]: s4 = pd.concat([s1 * 5, s3])

In [67]: pd.concat([s1, s4], axis=1)
Out[67]:
     0  1
a    0  0
b    1  5
f  NaN  5
g  NaN  6

In [68]: pd.concat([s1, s4], axis=1, join='inner')
Out[68]:
   0  1
a  0  0
b  1  5
```

You can even specify the axes to be used on the other axes with join_axes:

```
In [69]: pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])
Out[69]:
     0    1
a    0    0
c  NaN  NaN
b    1    5
e  NaN  NaN
```

One issue is that the concatenated pieces are not identifiable in the result. Suppose instead you wanted to create a hierarchical index on the concatenation axis. To do this, use the keys argument:

```
In [70]: result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])

In [71]: result
Out[71]:
one    a    0
       b    1
two    a    0
       b    1
three  f    5
       g    6

# Much more on the unstack function later
In [72]: result.unstack()
Out[72]:
```

```
        a    b    f    g
one     0    1  NaN  NaN
two     0    1  NaN  NaN
three NaN  NaN    5    6
```

In the case of combining Series along `axis=1`, the `keys` become the DataFrame column headers:

```
In [73]: pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])
Out[73]:
   one  two  three
a    0  NaN    NaN
b    1  NaN    NaN
c  NaN    2    NaN
d  NaN    3    NaN
e  NaN    4    NaN
f  NaN  NaN      5
g  NaN  NaN      6
```

The same logic extends to DataFrame objects:

```
In [74]: df1 = DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'],
   ....:                 columns=['one', 'two'])

In [75]: df2 = DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],
   ....:                 columns=['three', 'four'])

In [76]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
Out[76]:
   level1       level2
      one  two   three  four
a       0    1       5     6
b       2    3     NaN   NaN
c       4    5       7     8
```

If you pass a dict of objects instead of a list, the dict's keys will be used for the `keys` option:

```
In [77]: pd.concat({'level1': df1, 'level2': df2}, axis=1)
Out[77]:
   level1       level2
      one  two   three  four
a       0    1       5     6
b       2    3     NaN   NaN
c       4    5       7     8
```

There are a couple of additional arguments governing how the hierarchical index is created (see Table 7-2):

```
In [78]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'],
   ....:           names=['upper', 'lower'])
Out[78]:
upper  level1       level2
lower     one  two   three  four
a           0    1       5     6
b           2    3     NaN   NaN
c           4    5       7     8
```

A last consideration concerns DataFrames in which the row index is not meaningful in the context of the analysis:

```
In [79]: df1 = DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])

In [80]: df2 = DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])

In [81]: df1
Out[81]:
          a         b         c         d
0 -0.204708  0.478943 -0.519439 -0.555730
1  1.965781  1.393406  0.092908  0.281746
2  0.769023  1.246435  1.007189 -1.296221

In [82]: df2
Out[82]:
          b         d         a
0  0.274992  0.228913  1.352917
1  0.886429 -2.001637 -0.371843
```

In this case, you can pass `ignore_index=True`:

```
In [83]: pd.concat([df1, df2], ignore_index=True)
Out[83]:
          a         b         c         d
0 -0.204708  0.478943 -0.519439 -0.555730
1  1.965781  1.393406  0.092908  0.281746
2  0.769023  1.246435  1.007189 -1.296221
3  1.352917  0.274992       NaN  0.228913
4 -0.371843  0.886429       NaN -2.001637
```

Table 7-2. concat function arguments

Argument	Description
`objs`	List or dict of pandas objects to be concatenated. The only required argument
`axis`	Axis to concatenate along; defaults to 0
`join`	One of `'inner'`, `'outer'`, defaulting to `'outer'`; whether to intersection (inner) or union (outer) together indexes along the other axes
`join_axes`	Specific indexes to use for the other n-1 axes instead of performing union/intersection logic
`keys`	Values to associate with objects being concatenated, forming a hierarchical index along the concatenation axis. Can either be a list or array of arbitrary values, an array of tuples, or a list of arrays (if multiple level arrays passed in `levels`)
`levels`	Specific indexes to use as hierarchical index level or levels if keys passed
`names`	Names for created hierarchical levels if `keys` and / or `levels` passed
`verify_integrity`	Check new axis in concatenated object for duplicates and raise exception if so. By default (`False`) allows duplicates
`ignore_index`	Do not preserve indexes along concatenation `axis`, instead producing a new `range(total_length)` index

Combining Data with Overlap

Another data combination situation can't be expressed as either a merge or concatenation operation. You may have two datasets whose indexes overlap in full or part. As a motivating example, consider NumPy's `where` function, which expressed a vectorized if-else:

```
In [84]: a = Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan],
   ....:            index=['f', 'e', 'd', 'c', 'b', 'a'])

In [85]: b = Series(np.arange(len(a), dtype=np.float64),
   ....:            index=['f', 'e', 'd', 'c', 'b', 'a'])

In [86]: b[-1] = np.nan

In [87]: a        In [88]: b        In [89]: np.where(pd.isnull(a), b, a)
Out[87]:          Out[88]:          Out[89]:
f    NaN          f     0           f    0.0
e    2.5          e     1           e    2.5
d    NaN          d     2           d    2.0
c    3.5          c     3           c    3.5
b    4.5          b     4           b    4.5
a    NaN          a   NaN           a    NaN
```

Series has a `combine_first` method, which performs the equivalent of this operation plus data alignment:

```
In [90]: b[:-2].combine_first(a[2:])
Out[90]:
a    NaN
b    4.5
c    3.0
d    2.0
e    1.0
f    0.0
```

With DataFrames, `combine_first` naturally does the same thing column by column, so you can think of it as "patching" missing data in the calling object with data from the object you pass:

```
In [91]: df1 = DataFrame({'a': [1., np.nan, 5., np.nan],
   ....:                  'b': [np.nan, 2., np.nan, 6.],
   ....:                  'c': range(2, 18, 4)})

In [92]: df2 = DataFrame({'a': [5., 4., np.nan, 3., 7.],
   ....:                  'b': [np.nan, 3., 4., 6., 8.]})

In [93]: df1.combine_first(df2)
Out[93]:
   a    b    c
0  1  NaN    2
1  4    2    6
2  5    4   10
3  3    6   14
4  7    8  NaN
```

Reshaping and Pivoting

There are a number of fundamental operations for rearranging tabular data. These are alternatingly referred to as *reshape* or *pivot* operations.

Reshaping with Hierarchical Indexing

Hierarchical indexing provides a consistent way to rearrange data in a DataFrame. There are two primary actions:

- `stack`: this "rotates" or pivots from the columns in the data to the rows
- `unstack`: this pivots from the rows into the columns

I'll illustrate these operations through a series of examples. Consider a small DataFrame with string arrays as row and column indexes:

```
In [94]: data = DataFrame(np.arange(6).reshape((2, 3)),
   ....:                  index=pd.Index(['Ohio', 'Colorado'], name='state'),
   ....:                  columns=pd.Index(['one', 'two', 'three'], name='number'))

In [95]: data
Out[95]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

Using the `stack` method on this data pivots the columns into the rows, producing a Series:

```
In [96]: result = data.stack()

In [97]: result
Out[97]:
state     number
Ohio      one       0
          two       1
          three     2
Colorado  one       3
          two       4
          three     5
```

From a hierarchically-indexed Series, you can rearrange the data back into a DataFrame with `unstack`:

```
In [98]: result.unstack()
Out[98]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

By default the innermost level is unstacked (same with `stack`). You can unstack a different level by passing a level number or name:

```
In [99]: result.unstack(0)        In [100]: result.unstack('state')
Out[99]:                          Out[100]:
state   Ohio  Colorado            state   Ohio  Colorado
number                            number
one        0         3            one        0         3
```

```
two         1          4            two         1          4
three       2          5            three       2          5
```

Unstacking might introduce missing data if all of the values in the level aren't found in each of the subgroups:

```
In [101]: s1 = Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])

In [102]: s2 = Series([4, 5, 6], index=['c', 'd', 'e'])

In [103]: data2 = pd.concat([s1, s2], keys=['one', 'two'])

In [104]: data2.unstack()
Out[104]:
      a    b  c  d    e
one   0    1  2  3  NaN
two NaN  NaN  4  5    6
```

Stacking filters out missing data by default, so the operation is easily invertible:

```
In [105]: data2.unstack().stack()       In [106]: data2.unstack().stack(dropna=False)
Out[105]:                               Out[106]:
one  a    0                             one  a      0
     b    1                                  b      1
     c    2                                  c      2
     d    3                                  d      3
two  c    4                                  e    NaN
     d    5                             two  a    NaN
     e    6                                  b    NaN
                                             c      4
                                             d      5
                                             e      6
```

When unstacking in a DataFrame, the level unstacked becomes the lowest level in the result:

```
In [107]: df = DataFrame({'left': result, 'right': result + 5},
   .....:                columns=pd.Index(['left', 'right'], name='side'))

In [108]: df
Out[108]:
side             left  right
state    number
Ohio     one        0      5
         two        1      6
         three      2      7
Colorado one        3      8
         two        4      9
         three      5     10

In [109]: df.unstack('state')                   In [110]: df.unstack('state').stack('side')
Out[109]:                                       Out[110]:
side     left            right                  state        Ohio  Colorado
state    Ohio  Colorado  Ohio  Colorado         number side
number                                          one    left     0         3
one         0         3     5         8                right    5         8
two         1         4     6         9         two    left     1         4
```

```
three       2          5      7        10            right     6         9
                                               three left      2         5
                                                     right     7        10
```

Pivoting "long" to "wide" Format

A common way to store multiple time series in databases and CSV is in so-called *long* or *stacked* format:

```
In [116]: ldata[:10]
Out[116]:
                 date     item     value
0 1959-03-31 00:00:00  realgdp  2710.349
1 1959-03-31 00:00:00     infl     0.000
2 1959-03-31 00:00:00    unemp     5.800
3 1959-06-30 00:00:00  realgdp  2778.801
4 1959-06-30 00:00:00     infl     2.340
5 1959-06-30 00:00:00    unemp     5.100
6 1959-09-30 00:00:00  realgdp  2775.488
7 1959-09-30 00:00:00     infl     2.740
8 1959-09-30 00:00:00    unemp     5.300
9 1959-12-31 00:00:00  realgdp  2785.204
```

Data is frequently stored this way in relational databases like MySQL as a fixed schema (column names and data types) allows the number of distinct values in the `item` column to increase or decrease as data is added or deleted in the table. In the above example `date` and `item` would usually be the primary keys (in relational database parlance), offering both relational integrity and easier joins and programmatic queries in many cases. The downside, of course, is that the data may not be easy to work with in long format; you might prefer to have a DataFrame containing one column per distinct `item` value indexed by timestamps in the `date` column. DataFrame's `pivot` method performs exactly this transformation:

```
In [117]: pivoted = ldata.pivot('date', 'item', 'value')

In [118]: pivoted.head()
Out[118]:
item        infl   realgdp  unemp
date
1959-03-31  0.00  2710.349    5.8
1959-06-30  2.34  2778.801    5.1
1959-09-30  2.74  2775.488    5.3
1959-12-31  0.27  2785.204    5.6
1960-03-31  2.31  2847.699    5.2
```

The first two values passed are the columns to be used as the row and column index, and finally an optional value column to fill the DataFrame. Suppose you had two value columns that you wanted to reshape simultaneously:

```
In [119]: ldata['value2'] = np.random.randn(len(ldata))

In [120]: ldata[:10]
Out[120]:
```

```
                 date     item     value    value2
0 1959-03-31 00:00:00  realgdp  2710.349  1.669025
1 1959-03-31 00:00:00     infl     0.000 -0.438570
2 1959-03-31 00:00:00    unemp     5.800 -0.539741
3 1959-06-30 00:00:00  realgdp  2778.801  0.476985
4 1959-06-30 00:00:00     infl     2.340  3.248944
5 1959-06-30 00:00:00    unemp     5.100 -1.021228
6 1959-09-30 00:00:00  realgdp  2775.488 -0.577087
7 1959-09-30 00:00:00     infl     2.740  0.124121
8 1959-09-30 00:00:00    unemp     5.300  0.302614
9 1959-12-31 00:00:00  realgdp  2785.204  0.523772
```

By omitting the last argument, you obtain a DataFrame with hierarchical columns:

```
In [121]: pivoted = ldata.pivot('date', 'item')

In [122]: pivoted[:5]
Out[122]:
           value                    value2
item        infl   realgdp  unemp     infl   realgdp     unemp
date
1959-03-31  0.00  2710.349    5.8 -0.438570  1.669025 -0.539741
1959-06-30  2.34  2778.801    5.1  3.248944  0.476985 -1.021228
1959-09-30  2.74  2775.488    5.3  0.124121 -0.577087  0.302614
1959-12-31  0.27  2785.204    5.6  0.000940  0.523772  1.343810
1960-03-31  2.31  2847.699    5.2 -0.831154 -0.713544 -2.370232

In [123]: pivoted['value'][:5]
Out[123]:
item        infl   realgdp  unemp
date
1959-03-31  0.00  2710.349    5.8
1959-06-30  2.34  2778.801    5.1
1959-09-30  2.74  2775.488    5.3
1959-12-31  0.27  2785.204    5.6
1960-03-31  2.31  2847.699    5.2
```

Note that `pivot` is just a shortcut for creating a hierarchical index using `set_index` and reshaping with `unstack`:

```
In [124]: unstacked = ldata.set_index(['date', 'item']).unstack('item')

In [125]: unstacked[:7]
Out[125]:
           value                    value2
item        infl   realgdp  unemp     infl   realgdp     unemp
date
1959-03-31  0.00  2710.349    5.8 -0.438570  1.669025 -0.539741
1959-06-30  2.34  2778.801    5.1  3.248944  0.476985 -1.021228
1959-09-30  2.74  2775.488    5.3  0.124121 -0.577087  0.302614
1959-12-31  0.27  2785.204    5.6  0.000940  0.523772  1.343810
1960-03-31  2.31  2847.699    5.2 -0.831154 -0.713544 -2.370232
1960-06-30  0.14  2834.390    5.2 -0.860757 -1.860761  0.560145
1960-09-30  2.70  2839.022    5.6  0.119827 -1.265934 -1.063512
```

Data Transformation

So far in this chapter we've been concerned with rearranging data. Filtering, cleaning, and other tranformations are another class of important operations.

Removing Duplicates

Duplicate rows may be found in a DataFrame for any number of reasons. Here is an example:

```
In [126]: data = DataFrame({'k1': ['one'] * 3 + ['two'] * 4,
   .....:                   'k2': [1, 1, 2, 3, 3, 4, 4]})

In [127]: data
Out[127]:
    k1  k2
0  one   1
1  one   1
2  one   2
3  two   3
4  two   3
5  two   4
6  two   4
```

The DataFrame method **duplicated** returns a boolean Series indicating whether each row is a duplicate or not:

```
In [128]: data.duplicated()
Out[128]:
0    False
1     True
2    False
3    False
4     True
5    False
6     True
```

Relatedly, **drop_duplicates** returns a DataFrame where the **duplicated** array is **True**:

```
In [129]: data.drop_duplicates()
Out[129]:
    k1  k2
0  one   1
2  one   2
3  two   3
5  two   4
```

Both of these methods by default consider all of the columns; alternatively you can specify any subset of them to detect duplicates. Suppose we had an additional column of values and wanted to filter duplicates only based on the `'k1'` column:

```
In [130]: data['v1'] = range(7)

In [131]: data.drop_duplicates(['k1'])
```

```
Out[131]:
    k1  k2  v1
0  one   1   0
3  two   3   3
```

`duplicated` and `drop_duplicates` by default keep the first observed value combination. Passing `take_last=True` will return the last one:

```
In [132]: data.drop_duplicates(['k1', 'k2'], take_last=True)
Out[132]:
    k1  k2  v1
1  one   1   1
2  one   2   2
4  two   3   4
6  two   4   6
```

Transforming Data Using a Function or Mapping

For many data sets, you may wish to perform some transformation based on the values in an array, Series, or column in a DataFrame. Consider the following hypothetical data collected about some kinds of meat:

```
In [133]: data = DataFrame({'food': ['bacon', 'pulled pork', 'bacon', 'Pastrami',
   .....:                            'corned beef', 'Bacon', 'pastrami', 'honey ham',
   .....:                            'nova lox'],
   .....:                   'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})

In [134]: data
Out[134]:
          food  ounces
0        bacon     4.0
1  pulled pork     3.0
2        bacon    12.0
3     Pastrami     6.0
4  corned beef     7.5
5        Bacon     8.0
6     pastrami     3.0
7    honey ham     5.0
8     nova lox     6.0
```

Suppose you wanted to add a column indicating the type of animal that each food came from. Let's write down a mapping of each distinct meat type to the kind of animal:

```
meat_to_animal = {
  'bacon': 'pig',
  'pulled pork': 'pig',
  'pastrami': 'cow',
  'corned beef': 'cow',
  'honey ham': 'pig',
  'nova lox': 'salmon'
}
```

The map method on a Series accepts a function or dict-like object containing a mapping, but here we have a small problem in that some of the meats above are capitalized and others are not. Thus, we also need to convert each value to lower case:

```
In [136]: data['animal'] = data['food'].map(str.lower).map(meat_to_animal)

In [137]: data
Out[137]:
          food  ounces  animal
0        bacon     4.0     pig
1  pulled pork     3.0     pig
2        bacon    12.0     pig
3     Pastrami     6.0     cow
4  corned beef     7.5     cow
5        Bacon     8.0     pig
6     pastrami     3.0     cow
7    honey ham     5.0     pig
8     nova lox     6.0  salmon
```

We could also have passed a function that does all the work:

```
In [138]: data['food'].map(lambda x: meat_to_animal[x.lower()])
Out[138]:
0       pig
1       pig
2       pig
3       cow
4       cow
5       pig
6       cow
7       pig
8    salmon
Name: food
```

Using map is a convenient way to perform element-wise transformations and other data cleaning-related operations.

Replacing Values

Filling in missing data with the fillna method can be thought of as a special case of more general value replacement. While map, as you've seen above, can be used to modify a subset of values in an object, replace provides a simpler and more flexible way to do so. Let's consider this Series:

```
In [139]: data = Series([1., -999., 2., -999., -1000., 3.])

In [140]: data
Out[140]:
0       1
1    -999
2       2
3    -999
4   -1000
5       3
```

The `-999` values might be sentinel values for missing data. To replace these with NA values that pandas understands, we can use `replace`, producing a new Series:

```
In [141]: data.replace(-999, np.nan)
Out[141]:
0       1
1     NaN
2       2
3     NaN
4   -1000
5       3
```

If you want to replace multiple values at once, you instead pass a list then the substitute value:

```
In [142]: data.replace([-999, -1000], np.nan)
Out[142]:
0     1
1   NaN
2     2
3   NaN
4   NaN
5     3
```

To use a different replacement for each value, pass a list of substitutes:

```
In [143]: data.replace([-999, -1000], [np.nan, 0])
Out[143]:
0     1
1   NaN
2     2
3   NaN
4     0
5     3
```

The argument passed can also be a dict:

```
In [144]: data.replace({-999: np.nan, -1000: 0})
Out[144]:
0     1
1   NaN
2     2
3   NaN
4     0
5     3
```

Renaming Axis Indexes

Like values in a Series, axis labels can be similarly transformed by a function or mapping of some form to produce new, differently labeled objects. The axes can also be modified in place without creating a new data structure. Here's a simple example:

```
In [145]: data = DataFrame(np.arange(12).reshape((3, 4)),
   .....:                  index=['Ohio', 'Colorado', 'New York'],
   .....:                  columns=['one', 'two', 'three', 'four'])
```

Like a Series, the axis indexes have a `map` method:

```
In [146]: data.index.map(str.upper)
Out[146]: array([OHIO, COLORADO, NEW YORK], dtype=object)
```

You can assign to `index`, modifying the DataFrame in place:

```
In [147]: data.index = data.index.map(str.upper)

In [148]: data
Out[148]:
          one  two  three  four
OHIO        0    1      2     3
COLORADO    4    5      6     7
NEW YORK    8    9     10    11
```

If you want to create a transformed version of a data set without modifying the original, a useful method is `rename`:

```
In [149]: data.rename(index=str.title, columns=str.upper)
Out[149]:
          ONE  TWO  THREE  FOUR
Ohio        0    1      2     3
Colorado    4    5      6     7
New York    8    9     10    11
```

Notably, `rename` can be used in conjunction with a dict-like object providing new values for a subset of the axis labels:

```
In [150]: data.rename(index={'OHIO': 'INDIANA'},
   .....:             columns={'three': 'peekaboo'})
Out[150]:
          one  two  peekaboo  four
INDIANA     0    1         2     3
COLORADO    4    5         6     7
NEW YORK    8    9        10    11
```

`rename` saves having to copy the DataFrame manually and assign to its `index` and `columns` attributes. Should you wish to modify a data set in place, pass `inplace=True`:

```
# Always returns a reference to a DataFrame
In [151]: _ = data.rename(index={'OHIO': 'INDIANA'}, inplace=True)

In [152]: data
Out[152]:
          one  two  three  four
INDIANA     0    1      2     3
COLORADO    4    5      6     7
NEW YORK    8    9     10    11
```

Discretization and Binning

Continuous data is often discretized or otherwised separated into “bins” for analysis. Suppose you have data about a group of people in a study, and you want to group them into discrete age buckets:

```
In [153]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

Let’s divide these into bins of 18 to 25, 26 to 35, 35 to 60, and finally 60 and older. To do so, you have to use `cut`, a function in pandas:

```
In [154]: bins = [18, 25, 35, 60, 100]

In [155]: cats = pd.cut(ages, bins)

In [156]: cats
Out[156]:
Categorical:
array([(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], (18, 25],
       (35, 60], (25, 35], (60, 100], (35, 60], (35, 60], (25, 35]], dtype=object)
Levels (4): Index([(18, 25], (25, 35], (35, 60], (60, 100]], dtype=object)
```

The object pandas returns is a special `Categorical` object. You can treat it like an array of strings indicating the bin name; internally it contains a `levels` array indicating the distinct category names along with a labeling for the `ages` data in the `labels` attribute:

```
In [157]: cats.labels
Out[157]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1])

In [158]: cats.levels
Out[158]: Index([(18, 25], (25, 35], (35, 60], (60, 100]], dtype=object)

In [159]: pd.value_counts(cats)
Out[159]:
(18, 25]     5
(35, 60]     3
(25, 35]     3
(60, 100]    1
```

Consistent with mathematical notation for intervals, a parenthesis means that the side is *open* while the square bracket means it is *closed* (inclusive). Which side is closed can be changed by passing `right=False`:

```
In [160]: pd.cut(ages, [18, 26, 36, 61, 100], right=False)
Out[160]:
Categorical:
array([[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), [18, 26),
       [36, 61), [26, 36), [61, 100), [36, 61), [36, 61), [26, 36)], dtype=object)
Levels (4): Index([[18, 26), [26, 36), [36, 61), [61, 100)], dtype=object)
```

You can also pass your own bin names by passing a list or array to the `labels` option:

```
In [161]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']

In [162]: pd.cut(ages, bins, labels=group_names)
Out[162]:
```

```
Categorical:
array([Youth, Youth, Youth, YoungAdult, Youth, Youth, MiddleAged,
       YoungAdult, Senior, MiddleAged, MiddleAged, YoungAdult], dtype=object)
Levels (4): Index([Youth, YoungAdult, MiddleAged, Senior], dtype=object)
```

If you pass cut a integer number of bins instead of explicit bin edges, it will compute equal-length bins based on the minimum and maximum values in the data. Consider the case of some uniformly distributed data chopped into fourths:

```
In [163]: data = np.random.rand(20)

In [164]: pd.cut(data, 4, precision=2)
Out[164]:
Categorical:
array([(0.45, 0.67], (0.23, 0.45], (0.0037, 0.23], (0.45, 0.67],
       (0.67, 0.9], (0.45, 0.67], (0.67, 0.9], (0.23, 0.45], (0.23, 0.45],
       (0.67, 0.9], (0.67, 0.9], (0.67, 0.9], (0.23, 0.45], (0.23, 0.45],
       (0.23, 0.45], (0.67, 0.9], (0.0037, 0.23], (0.0037, 0.23],
       (0.23, 0.45], (0.23, 0.45]], dtype=object)
Levels (4): Index([(0.0037, 0.23], (0.23, 0.45], (0.45, 0.67],
                   (0.67, 0.9]], dtype=object)
```

A closely related function, qcut, bins the data based on sample quantiles. Depending on the distribution of the data, using cut will not usually result in each bin having the same number of data points. Since qcut uses sample quantiles instead, by definition you will obtain roughly equal-size bins:

```
In [165]: data = np.random.randn(1000) # Normally distributed

In [166]: cats = pd.qcut(data, 4) # Cut into quartiles

In [167]: cats
Out[167]:
Categorical:
array([(-0.022, 0.641], [-3.745, -0.635], (0.641, 3.26], ...,
       (-0.635, -0.022], (0.641, 3.26], (-0.635, -0.022]], dtype=object)
Levels (4): Index([[-3.745, -0.635], (-0.635, -0.022], (-0.022, 0.641],
                   (0.641, 3.26]], dtype=object)

In [168]: pd.value_counts(cats)
Out[168]:
[-3.745, -0.635]    250
(0.641, 3.26]       250
(-0.635, -0.022]    250
(-0.022, 0.641]     250
```

Similar to cut you can pass your own quantiles (numbers between 0 and 1, inclusive):

```
In [169]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
Out[169]:
Categorical:
array([(-0.022, 1.302], (-1.266, -0.022], (-0.022, 1.302], ...,
       (-1.266, -0.022], (-0.022, 1.302], (-1.266, -0.022]], dtype=object)
Levels (4): Index([[-3.745, -1.266], (-1.266, -0.022], (-0.022, 1.302],
                   (1.302, 3.26]], dtype=object)
```

We'll return to cut and qcut later in the chapter on aggregation and group operations, as these discretization functions are especially useful for quantile and group analysis.

Detecting and Filtering Outliers

Filtering or transforming outliers is largely a matter of applying array operations. Consider a DataFrame with some normally distributed data:

```
In [170]: np.random.seed(12345)

In [171]: data = DataFrame(np.random.randn(1000, 4))

In [172]: data.describe()
Out[172]:
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean     -0.067684     0.067924     0.025598    -0.002298
std       0.998035     0.992106     1.006835     0.996794
min      -3.428254    -3.548824    -3.184377    -3.745356
25%      -0.774890    -0.591841    -0.641675    -0.644144
50%      -0.116401     0.101143     0.002073    -0.013611
75%       0.616366     0.780282     0.680391     0.654328
max       3.366626     2.653656     3.260383     3.927528
```

Suppose you wanted to find values in one of the columns exceeding three in magnitude:

```
In [173]: col = data[3]

In [174]: col[np.abs(col) > 3]
Out[174]:
97     3.927528
305   -3.399312
400   -3.745356
Name: 3
```

To select all rows having a value exceeding 3 or -3, you can use the any method on a boolean DataFrame:

```
In [175]: data[(np.abs(data) > 3).any(1)]
Out[175]:
            0         1         2         3
5   -0.539741  0.476985  3.248944 -1.021228
97  -0.774363  0.552936  0.106061  3.927528
102 -0.655054 -0.565230  3.176873  0.959533
305 -2.315555  0.457246 -0.025907 -3.399312
324  0.050188  1.951312  3.260383  0.963301
400  0.146326  0.508391 -0.196713 -3.745356
499 -0.293333 -0.242459 -3.056990  1.918403
523 -3.428254 -0.296336 -0.439938 -0.867165
586  0.275144  1.179227 -3.184377  1.369891
808 -0.362528 -3.548824  1.553205 -2.186301
900  3.366626 -2.372214  0.851010  1.332846
```

Values can just as easily be set based on these criteria. Here is code to cap values outside the interval -3 to 3:

```
In [176]: data[np.abs(data) > 3] = np.sign(data) * 3

In [177]: data.describe()
Out[177]:
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean     -0.067623     0.068473     0.025153    -0.002081
std       0.995485     0.990253     1.003977     0.989736
min      -3.000000    -3.000000    -3.000000    -3.000000
25%      -0.774890    -0.591841    -0.641675    -0.644144
50%      -0.116401     0.101143     0.002073    -0.013611
75%       0.616366     0.780282     0.680391     0.654328
max       3.000000     2.653656     3.000000     3.000000
```

The ufunc `np.sign` returns an array of 1 and -1 depending on the sign of the values.

Permutation and Random Sampling

Permuting (randomly reordering) a Series or the rows in a DataFrame is easy to do using the `numpy.random.permutation` function. Calling `permutation` with the length of the axis you want to permute produces an array of integers indicating the new ordering:

```
In [178]: df = DataFrame(np.arange(5 * 4).reshape(5, 4))

In [179]: sampler = np.random.permutation(5)

In [180]: sampler
Out[180]: array([1, 0, 2, 3, 4])
```

That array can then be used in `ix`-based indexing or the `take` function:

```
In [181]: df                 In [182]: df.take(sampler)
Out[181]:                    Out[182]:
    0   1   2   3                0   1   2   3
0   0   1   2   3            1   4   5   6   7
1   4   5   6   7            0   0   1   2   3
2   8   9  10  11            2   8   9  10  11
3  12  13  14  15            3  12  13  14  15
4  16  17  18  19            4  16  17  18  19
```

To select a random subset without replacement, one way is to slice off the first `k` elements of the array returned by `permutation`, where `k` is the desired subset size. There are much more efficient sampling-without-replacement algorithms, but this is an easy strategy that uses readily available tools:

```
In [183]: df.take(np.random.permutation(len(df))[:3])
Out[183]:
    0   1   2   3
1   4   5   6   7
3  12  13  14  15
4  16  17  18  19
```

To generate a sample *with* replacement, the fastest way is to use `np.random.randint` to draw random integers:

```
In [184]: bag = np.array([5, 7, -1, 6, 4])

In [185]: sampler = np.random.randint(0, len(bag), size=10)

In [186]: sampler
Out[186]: array([4, 4, 2, 2, 2, 0, 3, 0, 4, 1])

In [187]: draws = bag.take(sampler)

In [188]: draws
Out[188]: array([ 4,  4, -1, -1, -1,  5,  6,  5,  4,  7])
```

Computing Indicator/Dummy Variables

Another type of transformation for statistical modeling or machine learning applications is converting a categorical variable into a "dummy" or "indicator" matrix. If a column in a DataFrame has `k` distinct values, you would derive a matrix or DataFrame containing `k` columns containing all 1's and 0's. pandas has a `get_dummies` function for doing this, though devising one yourself is not difficult. Let's return to an earlier example DataFrame:

```
In [189]: df = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   .....:                 'data1': range(6)})

In [190]: pd.get_dummies(df['key'])
Out[190]:
   a  b  c
0  0  1  0
1  0  1  0
2  1  0  0
3  0  0  1
4  1  0  0
5  0  1  0
```

In some cases, you may want to add a prefix to the columns in the indicator DataFrame, which can then be merged with the other data. `get_dummies` has a prefix argument for doing just this:

```
In [191]: dummies = pd.get_dummies(df['key'], prefix='key')

In [192]: df_with_dummy = df[['data1']].join(dummies)

In [193]: df_with_dummy
Out[193]:
   data1  key_a  key_b  key_c
0      0      0      1      0
1      1      0      1      0
2      2      1      0      0
3      3      0      0      1
4      4      1      0      0
5      5      0      1      0
```

If a row in a DataFrame belongs to multiple categories, things are a bit more complicated. Let's return to the MovieLens 1M dataset from earlier in the book:

```
In [194]: mnames = ['movie_id', 'title', 'genres']

In [195]: movies = pd.read_table('ch07/movies.dat', sep='::', header=None,
   .....:                        names=mnames)

In [196]: movies[:10]
Out[196]:
   movie_id                               title                        genres
0         1                    Toy Story (1995)   Animation|Children's|Comedy
1         2                      Jumanji (1995)  Adventure|Children's|Fantasy
2         3             Grumpier Old Men (1995)                Comedy|Romance
3         4            Waiting to Exhale (1995)                  Comedy|Drama
4         5  Father of the Bride Part II (1995)                        Comedy
5         6                         Heat (1995)         Action|Crime|Thriller
6         7                      Sabrina (1995)                Comedy|Romance
7         8                 Tom and Huck (1995)          Adventure|Children's
8         9                 Sudden Death (1995)                        Action
9        10                    GoldenEye (1995)     Action|Adventure|Thriller
```

Adding indicator variables for each genre requires a little bit of wrangling. First, we extract the list of unique genres in the dataset (using a nice `set.union` trick):

```
In [197]: genre_iter = (set(x.split('|')) for x in movies.genres)

In [198]: genres = sorted(set.union(*genre_iter))
```

Now, one way to construct the indicator DataFrame is to start with a DataFrame of all zeros:

```
In [199]: dummies = DataFrame(np.zeros((len(movies), len(genres))), columns=genres)
```

Now, iterate through each movie and set entries in each row of `dummies` to 1:

```
In [200]: for i, gen in enumerate(movies.genres):
   .....:     dummies.ix[i, gen.split('|')] = 1
```

Then, as above, you can combine this with `movies`:

```
In [201]: movies_windic = movies.join(dummies.add_prefix('Genre_'))

In [202]: movies_windic.ix[0]
Out[202]:
movie_id                                 1
title                     Toy Story (1995)
genres                 Animation|Children's|Comedy
Genre_Action                             0
Genre_Adventure                          0
Genre_Animation                          1
Genre_Children's                         1
Genre_Comedy                             1
Genre_Crime                              0
Genre_Documentary                        0
Genre_Drama                              0
Genre_Fantasy                            0
```

```
Genre_Film-Noir                                   0
Genre_Horror                                      0
Genre_Musical                                     0
Genre_Mystery                                     0
Genre_Romance                                     0
Genre_Sci-Fi                                      0
Genre_Thriller                                    0
Genre_War                                         0
Genre_Western                                     0
Name: 0
```

For much larger data, this method of constructing indicator variables with multiple membership is not especially speedy. A lower-level function leveraging the internals of the DataFrame could certainly be written.

A useful recipe for statistical applications is to combine `get_dummies` with a discretization function like `cut`:

```
In [204]: values = np.random.rand(10)

In [205]: values
Out[205]:
array([ 0.9296,  0.3164,  0.1839,  0.2046,  0.5677,  0.5955,  0.9645,
        0.6532,  0.7489,  0.6536])

In [206]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [207]: pd.get_dummies(pd.cut(values, bins))
Out[207]:
   (0, 0.2]  (0.2, 0.4]  (0.4, 0.6]  (0.6, 0.8]  (0.8, 1]
0         0           0           0           0         1
1         0           1           0           0         0
2         1           0           0           0         0
3         0           1           0           0         0
4         0           0           1           0         0
5         0           0           1           0         0
6         0           0           0           0         1
7         0           0           0           1         0
8         0           0           0           1         0
9         0           0           0           1         0
```

String Manipulation

Python has long been a popular data munging language in part due to its ease-of-use for string and text processing. Most text operations are made simple with the string object's built-in methods. For more complex pattern matching and text manipulations, regular expressions may be needed. pandas adds to the mix by enabling you to apply string and regular expressions concisely on whole arrays of data, additionally handling the annoyance of missing data.

String Object Methods

In many string munging and scripting applications, built-in string methods are sufficient. As an example, a comma-separated string can be broken into pieces with `split`:

```
In [208]: val = 'a,b,  guido'

In [209]: val.split(',')
Out[209]: ['a', 'b', '  guido']
```

`split` is often combined with `strip` to trim whitespace (including newlines):

```
In [210]: pieces = [x.strip() for x in val.split(',')]

In [211]: pieces
Out[211]: ['a', 'b', 'guido']
```

These substrings could be concatenated together with a two-colon delimiter using addition:

```
In [212]: first, second, third = pieces

In [213]: first + '::' + second + '::' + third
Out[213]: 'a::b::guido'
```

But, this isn't a practical generic method. A faster and more Pythonic way is to pass a list or tuple to the `join` method on the string `'::'`:

```
In [214]: '::'.join(pieces)
Out[214]: 'a::b::guido'
```

Other methods are concerned with locating substrings. Using Python's `in` keyword is the best way to detect a substring, though `index` and `find` can also be used:

```
In [215]: 'guido' in val
Out[215]: True

In [216]: val.index(',')        In [217]: val.find(':')
Out[216]: 1                     Out[217]: -1
```

Note the difference between `find` and `index` is that `index` raises an exception if the string isn't found (versus returning -1):

```
In [218]: val.index(':')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-218-280f8b2856ce> in <module>()
----> 1 val.index(':')
ValueError: substring not found
```

Relatedly, `count` returns the number of occurrences of a particular substring:

```
In [219]: val.count(',')
Out[219]: 2
```

`replace` will substitute occurrences of one pattern for another. This is commonly used to delete patterns, too, by passing an empty string:

```
In [220]: val.replace(',', '::')
Out[220]: 'a::b::  guido'

In [221]: val.replace(',', '')
Out[221]: 'ab  guido'
```

Regular expressions can also be used with many of these operations as you'll see below.

Table 7-3. Python built-in string methods

Argument	Description
`count`	Return the number of non-overlapping occurrences of substring in the string.
`endswith, startswith`	Returns True if string ends with suffix (starts with prefix).
`join`	Use string as delimiter for concatenating a sequence of other strings.
`index`	Return position of first character in substring if found in the string. Raises `ValueError` if not found.
`find`	Return position of first character of *first* occurrence of substring in the string. Like `index`, but returns -1 if not found.
`rfind`	Return position of first character of *last* occurrence of substring in the string. Returns -1 if not found.
`replace`	Replace occurrences of string with another string.
`strip, rstrip, lstrip`	Trim whitespace, including newlines; equivalent to `x.strip()` (and `rstrip`, `lstrip`, respectively) for each element.
`split`	Break string into list of substrings using passed delimiter.
`lower, upper`	Convert alphabet characters to lowercase or uppercase, respectively.
`ljust, rjust`	Left justify or right justify, respectively. Pad opposite side of string with spaces (or some other fill character) to return a string with a minimum width.

Regular expressions

Regular expressions provide a flexible way to search or match string patterns in text. A single expression, commonly called a *regex*, is a string formed according to the regular expression language. Python's built-in `re` module is responsible for applying regular expressions to strings; I'll give a number of examples of its use here.

> The art of writing regular expressions could be a chapter of its own and thus is outside the book's scope. There are many excellent tutorials and references on the internet, such as Zed Shaw's *Learn Regex The Hard Way* (*http://regex.learncodethehardway.org/book/*).

The `re` module functions fall into three categories: pattern matching, substitution, and splitting. Naturally these are all related; a regex describes a pattern to locate in the text, which can then be used for many purposes. Let's look at a simple example: suppose I wanted to split a string with a variable number of whitespace characters (tabs, spaces, and newlines). The regex describing one or more whitespace characters is `\s+`:

```
In [222]: import re

In [223]: text = "foo    bar\t baz  \tqux"

In [224]: re.split('\s+', text)
Out[224]: ['foo', 'bar', 'baz', 'qux']
```

When you call `re.split('\s+', text)`, the regular expression is first *compiled*, then its `split` method is called on the passed text. You can compile the regex yourself with `re.compile`, forming a reusable regex object:

```
In [225]: regex = re.compile('\s+')

In [226]: regex.split(text)
Out[226]: ['foo', 'bar', 'baz', 'qux']
```

If, instead, you wanted to get a list of all patterns matching the regex, you can use the `findall` method:

```
In [227]: regex.findall(text)
Out[227]: ['    ', '\t ', '  \t']
```

> To avoid unwanted escaping with \ in a regular expression, use *raw* string literals like `r'C:\x'` instead of the equivalent `'C:\\x'`.

Creating a regex object with `re.compile` is highly recommended if you intend to apply the same expression to many strings; doing so will save CPU cycles.

`match` and `search` are closely related to `findall`. While `findall` returns all matches in a string, `search` returns only the first match. More rigidly, `match` *only* matches at the beginning of the string. As a less trivial example, let's consider a block of text and a regular expression capable of identifying most email addresses:

```
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com
"""
pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'

# re.IGNORECASE makes the regex case-insensitive
regex = re.compile(pattern, flags=re.IGNORECASE)
```

Using `findall` on the text produces a list of the e-mail addresses:

```
In [229]: regex.findall(text)
Out[229]: ['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']
```

`search` returns a special match object for the first email address in the text. For the above regex, the match object can only tell us the start and end position of the pattern in the string:

```
In [230]: m = regex.search(text)

In [231]: m
Out[231]: <_sre.SRE_Match at 0x10a05de00>

In [232]: text[m.start():m.end()]
Out[232]: 'dave@google.com'
```

`regex.match` returns `None`, as it only will match if the pattern occurs at the start of the string:

```
In [233]: print regex.match(text)
None
```

Relatedly, `sub` will return a new string with occurrences of the pattern replaced by the a new string:

```
In [234]: print regex.sub('REDACTED', text)
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

Suppose you wanted to find email addresses and simultaneously segment each address into its 3 components: username, domain name, and domain suffix. To do this, put parentheses around the parts of the pattern to segment:

```
In [235]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

In [236]: regex = re.compile(pattern, flags=re.IGNORECASE)
```

A match object produced by this modified regex returns a tuple of the pattern components with its `groups` method:

```
In [237]: m = regex.match('wesm@bright.net')

In [238]: m.groups()
Out[238]: ('wesm', 'bright', 'net')
```

`findall` returns a list of tuples when the pattern has groups:

```
In [239]: regex.findall(text)
Out[239]:
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

`sub` also has access to groups in each match using special symbols like `\1, \2`, etc.:

```
In [240]: print regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text)
Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

There is much more to regular expressions in Python, most of which is outside the book's scope. To give you a flavor, one variation on the above email regex gives names to the match groups:

```
regex = re.compile(r"""
    (?P<username>[A-Z0-9._%+-]+)
    @
    (?P<domain>[A-Z0-9.-]+)
    \.
    (?P<suffix>[A-Z]{2,4})""", flags=re.IGNORECASE|re.VERBOSE)
```

The match object produced by such a regex can produce a handy dict with the specified group names:

```
In [242]: m = regex.match('wesm@bright.net')

In [243]: m.groupdict()
Out[243]: {'domain': 'bright', 'suffix': 'net', 'username': 'wesm'}
```

Table 7-4. Regular expression methods

Argument	Description
`findall, finditer`	Return all non-overlapping matching patterns in a string. `findall` returns a list of all patterns while `finditer` returns them one by one from an iterator.
`match`	Match pattern at start of string and optionally segment pattern components into groups. If the pattern matches, returns a match object, otherwise None.
`search`	Scan string for match to pattern; returning a match object if so. Unlike `match`, the match can be anywhere in the string as opposed to only at the beginning.
`split`	Break string into pieces at each occurrence of pattern.
`sub, subn`	Replace all (`sub`) or first n occurrences (`subn`) of pattern in string with replacement expression. Use symbols `\1, \2, ...` to refer to match group elements in the replacement string.

Vectorized string functions in pandas

Cleaning up a messy data set for analysis often requires a lot of string munging and regularization. To complicate matters, a column containing strings will sometimes have missing data:

```
In [244]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
   .....:         'Rob': 'rob@gmail.com', 'Wes': np.nan}

In [245]: data = Series(data)

In [246]: data                        In [247]: data.isnull()
Out[246]:                             Out[247]:
Dave     dave@google.com              Dave     False
Rob        rob@gmail.com              Rob      False
Steve    steve@gmail.com              Steve    False
Wes                  NaN              Wes       True
```

String and regular expression methods can be applied (passing a `lambda` or other function) to each value using `data.map`, but it will fail on the NA. To cope with this, Series has concise methods for string operations that skip NA values. These are accessed through Series's `str` attribute; for example, we could check whether each email address has `'gmail'` in it with `str.contains`:

```
In [248]: data.str.contains('gmail')
Out[248]:
Dave     False
Rob       True
Steve     True
Wes        NaN
```

Regular expressions can be used, too, along with any `re` options like `IGNORECASE`:

```
In [249]: pattern
Out[249]: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'

In [250]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[250]:
Dave     [('dave', 'google', 'com')]
Rob        [('rob', 'gmail', 'com')]
Steve    [('steve', 'gmail', 'com')]
Wes                              NaN
```

There are a couple of ways to do vectorized element retrieval. Either use `str.get` or index into the `str` attribute:

```
In [251]: matches = data.str.match(pattern, flags=re.IGNORECASE)

In [252]: matches
Out[252]:
Dave     ('dave', 'google', 'com')
Rob        ('rob', 'gmail', 'com')
Steve    ('steve', 'gmail', 'com')
Wes                            NaN

In [253]: matches.str.get(1)      In [254]: matches.str[0]
Out[253]:                         Out[254]:
Dave     google                   Dave      dave
Rob       gmail                   Rob        rob
Steve     gmail                   Steve    steve
Wes         NaN                   Wes        NaN
```

You can similarly slice strings using this syntax:

```
In [255]: data.str[:5]
Out[255]:
Dave     dave@
Rob      rob@g
Steve    steve
Wes        NaN
```

Table 7-5. Vectorized string methods

Method	Description
`cat`	Concatenate strings element-wise with optional delimiter
`contains`	Return boolean array if each string contains pattern/regex
`count`	Count occurrences of pattern
`endswith, startswith`	Equivalent to `x.endswith(pattern)` or `x.startswith(pattern)` for each element.
`findall`	Compute list of all occurrences of pattern/regex for each string
`get`	Index into each element (retrieve i-th element)
`join`	Join strings in each element of the Series with passed separator
`len`	Compute length of each string
`lower, upper`	Convert cases; equivalent to `x.lower()` or `x.upper()` for each element.
`match`	Use `re.match` with the passed regular expression on each element, returning matched groups as list.
`pad`	Add whitespace to left, right, or both sides of strings
`center`	Equivalent to `pad(side='both')`
`repeat`	Duplicate values; for example `s.str.repeat(3)` equivalent to `x * 3` for each string.
`replace`	Replace occurrences of pattern/regex with some other string
`slice`	Slice each string in the Series.
`split`	Split strings on delimiter or regular expression
`strip, rstrip, lstrip`	Trim whitespace, including newlines; equivalent to `x.strip()` (and `rstrip`, `lstrip`, respectively) for each element.

Example: USDA Food Database

The US Department of Agriculture makes available a database of food nutrient information. Ashley Williams, an English hacker, has made available a version of this database in JSON format (*http://ashleyw.co.uk/project/food-nutrient-database*). The records look like this:

```
{
  "id": 21441,
  "description": "KENTUCKY FRIED CHICKEN, Fried Chicken, EXTRA CRISPY,
Wing, meat and skin with breading",
  "tags": ["KFC"],
  "manufacturer": "Kentucky Fried Chicken",
  "group": "Fast Foods",
  "portions": [
    {
      "amount": 1,
      "unit": "wing, with skin",
      "grams": 68.0
    },
```

```
        ...
      ],
      "nutrients": [
        {
          "value": 20.8,
          "units": "g",
          "description": "Protein",
          "group": "Composition"
        },

        ...
      ]
    }
```

Each food has a number of identifying attributes along with two lists of nutrients and portion sizes. Having the data in this form is not particularly amenable for analysis, so we need to do some work to wrangle the data into a better form.

After downloading and extracting the data from the link above, you can load it into Python with any JSON library of your choosing. I'll use the built-in Python `json` module:

```
In [256]: import json

In [257]: db = json.load(open('ch07/foods-2011-10-03.json'))

In [258]: len(db)
Out[258]: 6636
```

Each entry in `db` is a dict containing all the data for a single food. The `'nutrients'` field is a list of dicts, one for each nutrient:

```
In [259]: db[0].keys()        In [260]: db[0]['nutrients'][0]
Out[259]:                     Out[260]:
[u'portions',                 {u'description': u'Protein',
 u'description',               u'group': u'Composition',
 u'tags',                      u'units': u'g',
 u'nutrients',                 u'value': 25.18}
 u'group',
 u'id',
 u'manufacturer']

In [261]: nutrients = DataFrame(db[0]['nutrients'])

In [262]: nutrients[:7]
Out[262]:
                   description        group units    value
0                      Protein  Composition     g    25.18
1            Total lipid (fat)  Composition     g    29.20
2   Carbohydrate, by difference  Composition     g     3.06
3                          Ash        Other     g     3.28
4                       Energy       Energy  kcal   376.00
5                        Water  Composition     g    39.28
6                       Energy       Energy    kJ  1573.00
```

When converting a list of dicts to a DataFrame, we can specify a list of fields to extract. We'll take the food names, group, id, and manufacturer:

```
In [263]: info_keys = ['description', 'group', 'id', 'manufacturer']

In [264]: info = DataFrame(db, columns=info_keys)

In [265]: info[:5]
Out[265]:
                          description                   group    id manufacturer
0                     Cheese, caraway  Dairy and Egg Products  1008
1                     Cheese, cheddar  Dairy and Egg Products  1009
2                        Cheese, edam  Dairy and Egg Products  1018
3                        Cheese, feta  Dairy and Egg Products  1019
4  Cheese, mozzarella, part skim milk  Dairy and Egg Products  1028

In [266]: info
Out[266]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 6636 entries, 0 to 6635
Data columns:
description     6636  non-null values
group           6636  non-null values
id              6636  non-null values
manufacturer    5195  non-null values
dtypes: int64(1), object(3)
```

You can see the distribution of food groups with `value_counts`:

```
In [267]: pd.value_counts(info.group)[:10]
Out[267]:
Vegetables and Vegetable Products    812
Beef Products                        618
Baked Products                       496
Breakfast Cereals                    403
Legumes and Legume Products          365
Fast Foods                           365
Lamb, Veal, and Game Products        345
Sweets                               341
Pork Products                        328
Fruits and Fruit Juices              328
```

Now, to do some analysis on all of the nutrient data, it's easiest to assemble the nutrients for each food into a single large table. To do so, we need to take several steps. First, I'll convert each list of food nutrients to a DataFrame, add a column for the food `id`, and append the DataFrame to a list. Then, these can be concatenated together with `concat`:

```
nutrients = []

for rec in db:
    fnuts = DataFrame(rec['nutrients'])
    fnuts['id'] = rec['id']
    nutrients.append(fnuts)

nutrients = pd.concat(nutrients, ignore_index=True)
```

If all goes well, `nutrients` should look like this:

```
In [269]: nutrients
Out[269]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 389355 entries, 0 to 389354
Data columns:
description    389355  non-null values
group          389355  non-null values
units          389355  non-null values
value          389355  non-null values
id             389355  non-null values
dtypes: float64(1), int64(1), object(3)
```

I noticed that, for whatever reason, there are duplicates in this DataFrame, so it makes things easier to drop them:

```
In [270]: nutrients.duplicated().sum()
Out[270]: 14179

In [271]: nutrients = nutrients.drop_duplicates()
```

Since `'group'` and `'description'` is in both DataFrame objects, we can rename them to make it clear what is what:

```
In [272]: col_mapping = {'description' : 'food',
   .....:                'group'       : 'fgroup'}

In [273]: info = info.rename(columns=col_mapping, copy=False)

In [274]: info
Out[274]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 6636 entries, 0 to 6635
Data columns:
food            6636  non-null values
fgroup          6636  non-null values
id              6636  non-null values
manufacturer    5195  non-null values
dtypes: int64(1), object(3)

In [275]: col_mapping = {'description' : 'nutrient',
   .....:                'group' : 'nutgroup'}

In [276]: nutrients = nutrients.rename(columns=col_mapping, copy=False)

In [277]: nutrients
Out[277]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 375176 entries, 0 to 389354
Data columns:
nutrient    375176  non-null values
nutgroup    375176  non-null values
units       375176  non-null values
value       375176  non-null values
```

```
id          375176  non-null values
dtypes: float64(1), int64(1), object(3)
```

With all of this done, we're ready to merge `info` with `nutrients`:

```
In [278]: ndata = pd.merge(nutrients, info, on='id', how='outer')

In [279]: ndata
Out[279]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 375176 entries, 0 to 375175
Data columns:
nutrient        375176  non-null values
nutgroup        375176  non-null values
units           375176  non-null values
value           375176  non-null values
id              375176  non-null values
food            375176  non-null values
fgroup          375176  non-null values
manufacturer    293054  non-null values
dtypes: float64(1), int64(1), object(6)

In [280]: ndata.ix[30000]
Out[280]:
nutrient                       Folic acid
nutgroup                         Vitamins
units                                 mcg
value                                   0
id                                   5658
food            Ostrich, top loin, cooked
fgroup                   Poultry Products
manufacturer
Name: 30000
```

The tools that you need to slice and dice, aggregate, and visualize this dataset will be explored in detail in the next two chapters, so after you get a handle on those methods you might return to this dataset. For example, we could a plot of median values by food group and nutrient type (see Figure 7-1):

```
In [281]: result = ndata.groupby(['nutrient', 'fgroup'])['value'].quantile(0.5)

In [282]: result['Zinc, Zn'].order().plot(kind='barh')
```

With a little cleverness, you can find which food is most dense in each nutrient:

```
by_nutrient = ndata.groupby(['nutgroup', 'nutrient'])

get_maximum = lambda x: x.xs(x.value.idxmax())
get_minimum = lambda x: x.xs(x.value.idxmin())

max_foods = by_nutrient.apply(get_maximum)[['value', 'food']]

# make the food a little smaller
max_foods.food = max_foods.food.str[:50]
```

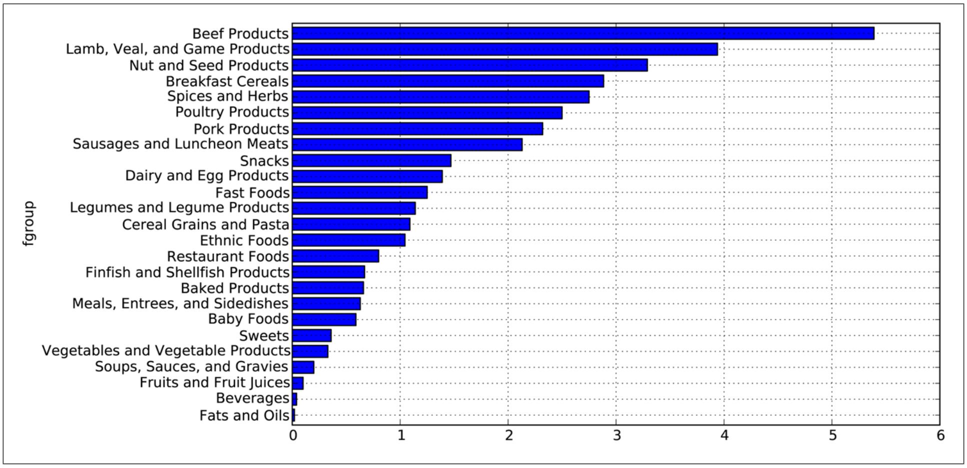

Figure 7-1. Median Zinc values by nutrient group

The resulting DataFrame is a bit too large to display in the book; here is just the `'Amino Acids'` nutrient group:

```
In [284]: max_foods.ix['Amino Acids']['food']
Out[284]:
nutrient
Alanine                          Gelatins, dry powder, unsweetened
Arginine                              Seeds, sesame flour, low-fat
Aspartic acid                                  Soy protein isolate
Cystine               Seeds, cottonseed flour, low fat (glandless)
Glutamic acid                                  Soy protein isolate
Glycine                          Gelatins, dry powder, unsweetened
Histidine               Whale, beluga, meat, dried (Alaska Native)
Hydroxyproline    KENTUCKY FRIED CHICKEN, Fried Chicken, ORIGINAL R
Isoleucine        Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Leucine           Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Lysine            Seal, bearded (Oogruk), meat, dried (Alaska Nativ
Methionine                   Fish, cod, Atlantic, dried and salted
Phenylalanine     Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Proline                          Gelatins, dry powder, unsweetened
Serine            Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Threonine         Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Tryptophan         Sea lion, Steller, meat with fat (Alaska Native)
Tyrosine          Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Valine            Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Name: food
```