Chapter 16. Data in a Box: Persistent Storage

> *It is a capital mistake to theorize before one has data.*
>
> —Arthur Conan Doyle

An active program accesses data stored in Random Access Memory, or RAM. RAM is very fast, but it is expensive and requires a constant supply of power; if the power goes out, all the data in memory is lost. Disk drives are slower than RAM but have more capacity, cost less, and retain data even after someone trips over the power cord. Thus, a huge amount of effort in computer systems has been devoted to making the best trade-offs between storing data on disk and RAM. As programmers, we need *persistence*: storing and retrieving data using nonvolatile media such as disks.

This chapter is all about the different flavors of data storage, each optimized for different purposes: flat files, structured files, and databases. File operations other than input and output are covered in Chapter 14.

A *record* is a term for one chunk of related data, consisting of individual *fields*.

Flat Text Files

The simplest persistence is a plain old flat file. This works well if your data has a very simple structure and you exchange all of it between disk and memory. Plain text data might be suitable for this treatment.

Padded Text Files

In this format, each field in a record has a fixed width, and is padded (usually with space characters) to that width in the file, giving each line (record) the same width. A programmer can use `seek()` to jump around the file and only read and write the records and fields that are needed.

Tabular Text Files

With simple text files, the only level of organization is the line. Sometimes, you want more structure than that. You might want to save data for your program to use later, or send data to another program.

There are many formats, and here's how you can distinguish them:

- A *separator*, or *delimiter*, character like tab (`'\t'`), comma (`','`), or vertical bar (`'|'`). This is an example of the comma-separated values (CSV) format.
- `'<'` and `'>'` around *tags*. Examples include XML and HTML.
- Punctuation. An example is JavaScript Object Notation (JSON).
- Indentation. An example is YAML (which is recursively defined as "YAML Ain't Markup Language").
- Miscellaneous, such as configuration files for programs.

Each of these structured file formats can be read and written by at least one Python module.

CSV

Delimited files are often used as an exchange format for spreadsheets and databases. You could read CSV files manually, a line at a time, splitting each line into fields at comma separators, and adding the results to data structures such as lists and dictionaries. But it's better to use the standard `csv` module, because parsing these files can get more complicated than you think. Following are a few important characteristics to keep in mind when working with CSV:

- Some have alternate delimiters besides a comma: `'|'` and `'\t'` (tab) are common.
- Some have *escape sequences*. If the delimiter character can occur within a field, the entire field might be surrounded by quote characters or preceded by some escape character.
- Files have different line-ending characters. Unix uses `'\n'`, Microsoft uses `'\r\n'`, and Apple used to use `'\r'` but now uses `'\n'`.
- The first line may contain column names.

First, we see how to read and write a list of rows, each containing a list of columns:

```
>>> import csv
>>> villains = [
...     ['Doctor', 'No'],
...     ['Rosa', 'Klebb'],
...     ['Mister', 'Big'],
...     ['Auric', 'Goldfinger'],
...     ['Ernst', 'Blofeld'],
...     ]
>>> with open('villains', 'wt') as fout:  # a context manager
...     csvout = csv.writer(fout)
...     csvout.writerows(villains)
```

This creates the file *villains* with these lines:

```
Doctor,No
Rosa,Klebb
Mister,Big
Auric,Goldfinger
Ernst,Blofeld
```

Now, we try to read it back in:

```
>>> import csv
>>> with open('villains', 'rt') as fin:  # context manager
...     cin = csv.reader(fin)
...     villains = [row for row in cin]  # a list comprehension
...
```

```
>>> print(villains)
[['Doctor', 'No'], ['Rosa', 'Klebb'], ['Mister', 'Big'],
['Auric', 'Goldfinger'], ['Ernst', 'Blofeld']]
```

We took advantage of the structure created by the `reader()` function. It created rows in the `cin` object that we could extract in a `for` loop.

Using `reader()` and `writer()` with their default options, the columns are separated by commas and the rows by line feeds.

The data can be a list of dictionaries rather than a list of lists. Let's read the *villains* file again, this time using the new `DictReader()` function and specifying the column names:

```
>>> import csv
>>> with open('villains', 'rt') as fin:
...     cin = csv.DictReader(fin, fieldnames=['first', 'last'])
...     villains = [row for row in cin]
...
>>> print(villains)
[OrderedDict([('first', 'Doctor'), ('last', 'No')]),
OrderedDict([('first', 'Rosa'), ('last', 'Klebb')]),
OrderedDict([('first', 'Mister'), ('last', 'Big')]),
OrderedDict([('first', 'Auric'), ('last', 'Goldfinger')]),
OrderedDict([('first', 'Ernst'), ('last', 'Blofeld')])]
```

That `OrderedDict` is there for compatibility with versions of Python before 3.6, when dictionaries kept their order by default.

Let's rewrite the CSV file by using the new `DictWriter()` function. We also call `writeheader()` to write an initial line of column names to the CSV file:

```
import csv
villains = [
    {'first': 'Doctor', 'last': 'No'},
    {'first': 'Rosa', 'last': 'Klebb'},
    {'first': 'Mister', 'last': 'Big'},
    {'first': 'Auric', 'last': 'Goldfinger'},
    {'first': 'Ernst', 'last': 'Blofeld'},
```

```
    ]
with open('villains.txt', 'wt') as fout:
    cout = csv.DictWriter(fout, ['first', 'last'])
    cout.writeheader()
    cout.writerows(villains)
```

That creates a *villains.csv* file with a header line (Example 16-1).

Example 16-1. villains.csv

```
first,last
Doctor,No
Rosa,Klebb
Mister,Big
Auric,Goldfinger
Ernst,Blofeld
```

Now, let's read it back. By omitting the `fieldnames` argument in the `DictReader()` call, we tell it to use the values in the first line of the file (`first,last`) as column labels and matching dictionary keys:

```
>>> import csv
>>> with open('villains.csv', 'rt') as fin:
...     cin = csv.DictReader(fin)
...     villains = [row for row in cin]
...
>>> print(villains)
[OrderedDict([('first', 'Doctor'), ('last', 'No')]),
OrderedDict([('first', 'Rosa'), ('last', 'Klebb')]),
OrderedDict([('first', 'Mister'), ('last', 'Big')]),
OrderedDict([('first', 'Auric'), ('last', 'Goldfinger')]),
OrderedDict([('first', 'Ernst'), ('last', 'Blofeld')])]
```

XML

Delimited files convey only two dimensions: rows (lines) and columns (fields within a line). If you want to exchange data structures among programs, you need a way to encode hierarchies, sequences, sets, and other structures as text.

XML is a prominent *markup* format that does this. It uses *tags* to delimit data, as in this sample *menu.xml* file:

```
<?xml version="1.0"?>
<menu>
  <breakfast hours="7-11">
    <item price="$6.00">breakfast burritos</item>
    <item price="$4.00">pancakes</item>
  </breakfast>
  <lunch hours="11-3">
    <item price="$5.00">hamburger</item>
  </lunch>
  <dinner hours="3-10">
    <item price="8.00">spaghetti</item>
  </dinner>
</menu>
```

Following are a few important characteristics of XML:

- Tags begin with a `<` character. The tags in this sample were `menu`, `breakfast`, `lunch`, `dinner`, and `item`.
- Whitespace is ignored.
- Usually a *start tag* such as `<menu>` is followed by other content and then a final matching *end tag* such as `</menu>`.
- Tags can *nest* within other tags to any level. In this example, `item` tags are children of the `breakfast`, `lunch`, and `dinner` tags; they, in turn, are children of `menu`.
- Optional *attributes* can occur within the start tag. In this example, `price` is an attribute of `item`.
- Tags can contain *values*. In this example, each `item` has a value, such as `pancakes` for the second breakfast item.
- If a tag named `thing` has no values or children, it can be expressed as the single tag by including a forward slash just before the closing angle bracket, such as `<thing/>`, rather than a start and end tag, like `<thing></thing>`.
- The choice of where to put data—attributes, values, child tags—is somewhat arbitrary. For instance, we could have written the last `item` tag as `<item price="$8.00" food="spaghetti"/>`.

XML is often used for data *feeds* and *messages* and has subformats like RSS and Atom. Some industries have many specialized XML formats, such as the finance field.

XML's über-flexibility has inspired multiple Python libraries that differ in approach and capabilities.

The simplest way to parse XML in Python is by using the standard `ElementTree` module. Here's a little program to parse the *menu.xml* file and print some tags and attributes:

```
>>> import xml.etree.ElementTree as et
>>> tree = et.ElementTree(file='menu.xml')
>>> root = tree.getroot()
>>> root.tag
'menu'
>>> for child in root:
...     print('tag:', child.tag, 'attributes:', child.attrib)
...     for grandchild in child:
...         print('\ttag:', grandchild.tag, 'attributes:', grandchild.attrib)
...
tag: breakfast attributes: {'hours': '7-11'}
    tag: item attributes: {'price': '$6.00'}
    tag: item attributes: {'price': '$4.00'}
tag: lunch attributes: {'hours': '11-3'}
    tag: item attributes: {'price': '$5.00'}
tag: dinner attributes: {'hours': '3-10'}
    tag: item attributes: {'price': '8.00'}
>>> len(root)     # number of menu sections
3
>>> len(root[0])  # number of breakfast items
2
```

For each element in the nested lists, `tag` is the tag string and `attrib` is a dictionary of its attributes. `ElementTree` has many other ways of searching XML-derived data, modifying it, and even writing XML files. The `ElementTree` documentation has the details.

Other standard Python XML libraries include the following:

`xml.dom`

The Document Object Model (DOM), familiar to JavaScript developers, represents web documents as hierarchical structures. This module loads the entire XML file into memory and lets you access all the pieces equally.

`xml.sax`

Simple API for XML, or SAX, parses XML on the fly, so it does not have to load everything into memory at once. Therefore, it can be a good choice if you need to process very large streams of XML.

An XML Security Note

You can use all the formats described in this chapter to save objects to files and read them back again. It's possible to exploit this process and cause security problems.

For example, the following XML snippet from the billion laughs Wikipedia page defines 10 nested entities, each expanding the lower level 10 times for a total expansion of one billion:

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol
 <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&
 <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&
 <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&
 <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&
 <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&
 <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&
 <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&
 <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&
]>
<lolz>&lol9;</lolz>
```

The bad news: billion laughs would blow up all of the XML libraries mentioned in the previous sections. Defused XML lists this attack and others,

along with the vulnerability of Python libraries.

The link shows how to change the settings for many of the libraries to avoid these problems. Also, you can use the `defusedxml` library as a security front-end for the other libraries:

```
>>> # insecure:
>>> from xml.etree.ElementTree import parse
>>> et = parse(xmlfile)
>>> # protected:
>>> from defusedxml.ElementTree import parse
>>> et = parse(xmlfile)
```

The standard Python site also has its own page on XML vulnerabilities.

HTML

Gigagobs of data are saved as Hypertext Markup Language (HTML), the basic document format of the web. The problem is that much of it doesn't follow the HTML rules, which can make it difficult to parse. HTML is a better display format than a data interchange format. Because this chapter is intended to describe fairly well-defined data formats, I've separated out the discussion about HTML to Chapter 18.

JSON

JavaScript Object Notation (JSON) has become a very popular data interchange format, beyond its JavaScript origins. The JSON format is a subset of JavaScript, and often legal Python syntax, as well. Its close fit to Python makes it a good choice for data interchange among programs. You'll see many examples of JSON for web development in Chapter 18.

Unlike the variety of XML modules, there's one main JSON module, with the unforgettable name `json`. This program encodes (dumps) data to a JSON string and decodes (loads) a JSON string back to data. In this next example, let's build a Python data structure containing the data from the earlier XML example:

```
>>> menu = \
... {
... "breakfast": {
...         "hours": "7-11",
...         "items": {
...                 "breakfast burritos": "$6.00",
...                 "pancakes": "$4.00"
...                 }
...         },
... "lunch" : {
...         "hours": "11-3",
...         "items": {
...                 "hamburger": "$5.00"
...                 }
...         },
... "dinner": {
...         "hours": "3-10",
...         "items": {
...                 "spaghetti": "$8.00"
...                 }
...         }
... }
.
```

Next, encode the data structure (`menu`) to a JSON string (`menu_json`) by using `dumps()`:

```
>>> import json
>>> menu_json = json.dumps(menu)
>>> menu_json
'{"dinner": {"items": {"spaghetti": "$8.00"}, "hours": "3-10"},
"lunch": {"items": {"hamburger": "$5.00"}, "hours": "11-3"},
"breakfast": {"items": {"breakfast burritos": "$6.00", "pancakes":
"$4.00"}, "hours": "7-11"}}'
```

And now, let's turn the JSON string `menu_json` back into a Python data structure (`menu2`) by using `loads()`:

```
>>> menu2 = json.loads(menu_json)
>>> menu2
```

```
{'breakfast': {'items': {'breakfast burritos': '$6.00', 'pancakes':
'$4.00'}, 'hours': '7-11'}, 'lunch': {'items': {'hamburger': '$5.00'},
'hours': '11-3'}, 'dinner': {'items': {'spaghetti': '$8.00'}, 'hours': '3-10'}}
```

`menu` and `menu2` are both dictionaries with the same keys and values.

You might get an exception while trying to encode or decode some objects, including objects such as `datetime` (covered in detail in Chapter 13), as demonstrated here:

```
>>> import datetime
>>> import json
>>> now = datetime.datetime.utcnow()
>>> now
datetime.datetime(2013, 2, 22, 3, 49, 27, 483336)
>>> json.dumps(now)
Traceback (most recent call last):
# ... (deleted stack trace to save trees)
TypeError: datetime.datetime(2013, 2, 22, 3, 49, 27, 483336)
  is not JSON serializable
>>>
```

This can happen because the JSON standard does not define date or time types; it expects you to define how to handle them. You could convert the `datetime` to something JSON understands, such as a string or an *epoch* value (see Chapter 13):

```
>>> now_str = str(now)
>>> json.dumps(now_str)
'"2013-02-22 03:49:27.483336"'
>>> from time import mktime
>>> now_epoch = int(mktime(now.timetuple()))
>>> json.dumps(now_epoch)
'1361526567'
```

If the `datetime` value could occur in the middle of normally converted data types, it might be annoying to make these special conversions. You can modify how JSON is encoded by using inheritance, which is described in Chapter 10. Python's JSON documentation gives an example of this for com-

plex numbers, which also makes JSON play dead. Let's modify it for `datetime`:

```
>>> import datetime
>>> now = datetime.datetime.utcnow()
>>> class DTEncoder(json.JSONEncoder):
...     def default(self, obj):
...         # isinstance() checks the type of obj
...         if isinstance(obj, datetime.datetime):
...             return int(mktime(obj.timetuple()))
...         # else it's something the normal decoder knows:
...         return json.JSONEncoder.default(self, obj)
...
>>> json.dumps(now, cls=DTEncoder)
'1361526567'
```

The new class `DTEncoder` is a subclass, or child class, of `JSONEncoder`. We need to override its only `default()` method to add `datetime` handling. Inheritance ensures that everything else will be handled by the parent class.

The `isinstance()` function checks whether the object `obj` is of the class `datetime.datetime`. Because everything in Python is an object, `isinstance()` works everywhere:

```
>>> import datetime
>>> now = datetime.datetime.utcnow()
>>> type(now)
<class 'datetime.datetime'>
>>> isinstance(now, datetime.datetime)
True
>>> type(234)
<class 'int'>
>>> isinstance(234, int)
True
>>> type('hey')
<class 'str'>
>>> isinstance('hey', str)
True
```

NOTE

For JSON and other structured text formats, you can load from a file into data structures without knowing anything about the structures ahead of time. Then, you can walk through the structures by using `isinstance()` and type-appropriate methods to examine their values. For example, if one of the items is a dictionary, you can extract contents through `keys()`, `values()`, and `items()`.

After making you do it the hard way, it turns out that there's an even easier way to convert `datetime` objects to JSON:

```
>>> import datetime
>>> import json
>>> now = datetime.datetime.utcnow()
>>> json.dumps(now, default=str)
'"2019-04-17 21:54:43.617337"'
```

That `default=str` tells `json.dumps()` to apply the `str()` conversion function for data types that it doesn't understand. This works because the definition of the `datetime.datetime` class includes a `__str__()` method.

YAML

Similar to JSON, YAML has keys and values, but handles more data types such as dates and times. The standard Python library does not yet include YAML handling, so you need to install a third-party library named `yaml` to manipulate it. `((("dump() function")))(("load() function")))load()` converts a YAML string to Python data, whereas `dump()` does the opposite.

The following YAML file, *mcintyre.yaml*, contains information on the Canadian poet James McIntyre, including two of his poems:

```
name:
  first: James
  last: McIntyre
dates:
  birth: 1828-05-25
  death: 1906-03-31
```

```
details:
  bearded: true
  themes: [cheese, Canada]
books:
  url: http://www.gutenberg.org/files/36068/36068-h/36068-h.htm
poems:
  - title: 'Motto'
    text: |
      Politeness, perseverance and pluck,
      To their possessor will bring good luck.
  - title: 'Canadian Charms'
    text: |
      Here industry is not in vain,
      For we have bounteous crops of grain,
      And you behold on every field
      Of grass and roots abundant yield,
      But after all the greatest charm
      Is the snug home upon the farm,
      And stone walls now keep cattle warm.
```

Values such as `true`, `false`, `on`, and `off` are converted to Python booleans. Integers and strings are converted to their Python equivalents. Other syntax creates lists and dictionaries:

```
>>> import yaml
>>> with open('mcintyre.yaml', 'rt') as fin:
>>>     text = fin.read()
>>> data = yaml.load(text)
>>> data['details']
{'themes': ['cheese', 'Canada'], 'bearded': True}
>>> len(data['poems'])
2
```

The data structures that are created match those in the YAML file, which in this case are more than one level deep in places. You can get the title of the second poem with this dict/list/dict reference:

```
>>> data['poems'][1]['title']
'Canadian Charms'
```

WARNING

PyYAML can load Python objects from strings, and this is dangerous. Use `safe_load()` instead of `load()` if you're importing YAML that you don't trust. Better yet, *always* use `safe_load()`. Read Ned Batchelder's blog post "War is Peace" for a description of how unprotected YAML loading compromised the Ruby on Rails platform.

Tablib

After reading all of the previous sections, there's one third-party package that lets you import, export, and edit tabular data in CSV, JSON, *or* YAML format,[1] as well as Microsoft Excel, Pandas DataFrame, and a few others. You install it with the familiar refrain (`pip install tablib`), and peek at the docs.

Pandas

This is as good place as any to introduce pandas—a Python library for structured data. It's an excellent tool for handling real-life data issues:

- Read and write many text and binary file formats:
  - Text, with fields separated by commas (CSV), tabs (TSV), or other characters
  - Fixed-width text
  - Excel
  - JSON
  - HTML tables
  - SQL
  - HDF5
  - and others.
- Group, split, merge, index, slice, sort, select, label
- Convert data types
- Change size or shape
- Handle missing data
- Generate random values
- Manage time series

The read functions return a `DataFrame` object, Pandas' standard representation for two-dimensional data (rows and columns). It's similar in some ways to a spreadsheet or a relational database table. Its one-dimensional little brother is a `Series`.

Example 16-2 demonstrates a simple application that reads our *villains.csv* file from Example 16-1.

Example 16-2. Read CSV with Pandas

```
>>> import pandas
>>>
>>> data = pandas.read_csv('villains.csv')
>>> print(data)
    first        last
0  Doctor          No
1    Rosa       Klebb
2  Mister         Big
3   Auric  Goldfinger
4   Ernst     Blofeld
```

The variable `data` just shown is a `DataFrame`. which has many more tricks than a basic Python dictionary. It's especially useful for heavy numeric work with NumPy, and data preparation for machine learning.

Refer to the "Getting Started" section of the documentation for Pandas' features, and "10 Minutes to Pandas" for working examples.

Let's use Pandas for a little calendar example—make a list of the first day of the first three months in 2019:

```
>>> import pandas
>>> dates = pandas.date_range('2019-01-01', periods=3, freq='MS')
>>> dates
DatetimeIndex(['2019-01-01', '2019-02-01', '2019-03-01'],
  dtype='datetime64[ns]', freq='MS')
```

You could write something that does this, using the time and date functions described in Chapter 13, but it would be a lot more work—especially debug-

ging (dates and times are frustrating). Pandas also handles many special date/time details, like business months and years.

Pandas will appear again later when I talk about mapping ("Geopandas") and scientific applications ("Pandas").

Configuration Files

Most programs offer various *options* or *settings*. Dynamic ones can be provided as program arguments, but long-lasting ones need to be kept somewhere. The temptation to define your own quick-and-dirty *config file* format is strong—but resist it. It often turns out to be dirty, but not so quick. You need to maintain both the writer program and the reader program (sometimes called a *parser*). There are good alternatives that you can just drop into your program, including those in the previous sections.

Here, we'll use the standard `configparser` module, which handles Windows-style *.ini* files. Such files have sections of *key* = *value* definitions. Here's a minimal *settings.cfg* file:

```
[english]
greeting = Hello

[french]
greeting = Bonjour

[files]
home = /usr/local
# simple interpolation:
bin = %(home)s/bin
```

Here's the code to read it into Python data structures:

```
>>> import configparser
>>> cfg = configparser.ConfigParser()
>>> cfg.read('settings.cfg')
['settings.cfg']
>>> cfg
```

```
<configparser.ConfigParser object at 0x1006be4d0>
>>> cfg['french']
<Section: french>
>>> cfg['french']['greeting']
'Bonjour'
>>> cfg['files']['bin']
'/usr/local/bin'
```

Other options are available, including fancier interpolation. See the `configparser` documentation. If you need deeper nesting than two levels, try YAML or JSON.

Binary Files

Some file formats were designed to store particular data structures but are neither relational nor NoSQL databases. The sections that follow present some of them.

Padded Binary Files and Memory Mapping

These are similar to padded text files, but the contents may be binary, and the padding byte may be `\x00` instead of a space character. Each record has a fixed size, as does each field within a record. This makes it simpler to `seek()` throughout the file for the desired records and fields. Every operation on the data is manual, so this approach tends to be used only in very low-level (e.g., close to the hardware) situations.

Data in this form can be *memory mapped* with the standard `mmap` library. See some examples, and the standard documentation.

Spreadsheets

Spreadsheets, notably Microsoft Excel, are widespread binary data formats. If you can save your spreadsheet to a CSV file, you can read it by using the standard `csv` module that was described earlier. This will work for a binary `xls` file, `xlrd`, or `tablib` (mentioned earlier at "Tablib").

HDF5

HDF5 is a binary data format for multidimensional or hierarchical numeric data. It's used mainly in science, where fast random access to large datasets (gigabytes to terabytes) is a common requirement. Even though HDF5 could be a good alternative to databases in some cases, for some reason HDF5 is almost unknown in the business world. It's best suited to *WORM* (write once/read many) applications for which database protection against conflicting writes is not needed. Here are some modules that you might find useful:

- `h5py` is a full-featured low-level interface. Read the documentation and code.
- `PyTables` is a bit higher-level, with database-like features. Read the documentation and code.

Both of these are discussed in terms of scientific applications of Python in Chapter 22. I'm mentioning HDF5 here in case you have a need to store and retrieve large amounts of data and are willing to consider something outside the box as well as the usual database solutions. A good example is the Million Song dataset, which has downloadable song data in HDF5 and SQLite formats.

TileDB

A recent successor to HDF5 for dense or spare array storage is TileDB. Install the Python interface (which includes the TileDB library itself) by running `pip install tiledb`. This is aimed at scientific data and applications.

Relational Databases

Relational databases are only about 40 years old but are ubiquitous in the computing world. You'll almost certainly have to deal with them at one time or another. When you do, you'll appreciate what they provide:

- Access to data by multiple simultaneous users
- Protection from corruption by those users

- Efficient methods to store and retrieve the data
- Data defined by *schemas* and limited by *constraints*
- *Joins* to find relationships across diverse types of data
- A declarative (rather than imperative) query language: *SQL* (Structured Query Language)

These are called *relational* because they show relationships among different kinds of data in the form of rectangular *tables*. For instance, in our menu example earlier, there is a relationship between each item and its price.

A table is a rectangular grid of *columns* (data fields) and *rows* (individual data records), similar to a spreadsheet. The intersection of a row and column is a table *cell*. To create a table, name it and specify the order, names, and types of its columns. Each row has the same columns, although a column may be defined to allow missing data (called *nulls*) in cells. In the menu example, you could create a table with one row for each item being sold. Each item has the same columns, including one for the price.

A column or group of columns is usually the table's *primary key;* its values must be unique in the table. This prevents adding the same data to the table more than once. This key is *indexed* for fast lookups during queries. An index works a little like a book index, making it fast to find a particular row.

Each table lives within a parent *database*, like a file within a directory. Two levels of hierarchy help keep things organized a little better.

NOTE

Yes, the word *database* is used in multiple ways: as the server, the table container, and the data stored therein. If you'll be referring to all of them at the same time, it might help to call them *database server*, *database*, and *data*.

If you want to find rows by some nonkey column value, define a *secondary index* on that column. Otherwise, the database server must perform a *table scan*—a brute-force search of every row for matching column values.

Tables can be related to each other with *foreign keys*, and column values can be constrained to these keys.

SQL

SQL is not an API or a protocol, but a declarative *language*: you say *what* you want rather than *how* to do it. It's the universal language of relational databases. SQL queries are text strings sent by a client to the database server, which in turn figures out what to do with them.

There have been various SQL standard definitions, and all database vendors have added their own tweaks and extensions, resulting in many SQL *dialects*. If you store your data in a relational database, SQL gives you some portability. Still, dialect and operational differences can make it difficult to move your data to another type of database. There are two main categories of SQL statements:

DDL (data definition language)
: Handles creation, deletion, constraints, and permissions for tables, databases, and users.

DML (data manipulation language)
: Handles data insertions, selects, updates, and deletions.

Table 16-1 lists the basic SQL DDL commands.

Table 16-1. Basic SQL DDL commands

Operation	SQL pattern	SQL example
Create a database	`CREATE DATABASE` *dbname*	`CREATE DATABASE d`
Select current database	`USE` *dbname*	`USE d`
Delete a database and its tables	`DROP DATABASE` *dbname*	`DROP DATABASE d`
Create a table	`CREATE TABLE` *tbname* `(` *coldefs* `)`	`CREATE TABLE t (id INT, count INT)`
Delete a table	`DROP TABLE` *tbname*	`DROP TABLE t`
Remove all rows from a table	`TRUNCATE TABLE` *tbname*	`TRUNCATE TABLE t`

NOTE

Why all the CAPITAL LETTERS? SQL is case-insensitive, but it's a tradition (don't ask me why) to SHOUT its keywords in code examples to distinguish them from column names.

The main DML operations of a relational database are often known by the acronym CRUD:

- *C*reate by using the SQL `INSERT` statement
- *R*ead by using `SELECT`
- *U*pdate by using `UPDATE`
- *D*elete by using `DELETE`

Table 16-2 looks at the commands available for SQL DML.

Table 16-2. Basic SQL DML commands

Operation	SQL pattern	SQL example
Add a row	`INSERT INTO` *tbname* `VALUES(` ... `)`	`INSERT INTO t VALUES (7, 40)`
Select all rows and columns	`SELECT * FROM` *tbname*	`SELECT * FROM t`
Select all rows, some columns	`SELECT` *cols* `FROM` *tbname*	`SELECT id, count FROM t`
Select some rows, some columns	`SELECT` *cols* `FROM` *tbname* `WHERE` *condition*	`SELECT id, count from t WHERE count > 5 AND id = 9`
Change some rows in a column	`UPDATE` *tbname* `SET` *col* `=` *value* `WHERE` *condition*	`UPDATE t SET count=3 WHERE id=5`
Delete some rows	`DELETE FROM` *tbname* `WHERE` *condition*	`DELETE FROM t WHERE count <= 10 OR id = 16`

DB-API

An application programming interface (API) is a set of functions that you can call to get access to some service. DB-API is Python's standard API for accessing relational databases. Using it, you can write a single program that works with multiple kinds of relational databases instead of writing a separate program for each one. It's similar to Java's JDBC or Perl's dbi.

Its main functions are the following:

`connect()`
: Make a connection to the database; this can include arguments such as username, password, server address, and others.

`cursor()`
: Create a *cursor* object to manage queries.

`execute()` *and* `executemany()`
: Run one or more SQL commands against the database.

`fetchone()`, `fetchmany()`, *and* `fetchall()`
: Get the results from `execute()`.

The Python database modules in the coming sections conform to DB-API, often with extensions and some differences in details.

SQLite

SQLite is a good, light, open source relational database. It's implemented as a standard Python library, and stores databases in normal files. These files are portable across machines and operating systems, making SQLite a very portable solution for simple relational database applications. It isn't as full featured as MySQL or PostgreSQL, but it does support SQL, and it manages multiple simultaneous users. Web browsers, smartphones, and other applications use SQLite as an embedded database.

You begin with a `connect()` to the local SQLite database file that you want to use or create. This file is the equivalent of the directory-like *database* that parents tables in other servers. The special string `':memory:'` creates the database in memory only; this is fast and useful for testing but will lose data when your program terminates or if your computer goes down.

For the next example, let's make a database called `enterprise.db` and the table `zoo` to manage our thriving roadside petting zoo business. The table columns are as follows:

`critter`
: A variable length string, and our primary key.

`count`
: An integer count of our current inventory for this animal.

`damages`
: The dollar amount of our current losses from animal–human interactions.

```
>>> import sqlite3
>>> conn = sqlite3.connect('enterprise.db')
>>> curs = conn.cursor()
>>> curs.execute('''CREATE TABLE zoo
    (critter VARCHAR(20) PRIMARY KEY,
     count INT,
     damages FLOAT)''')
<sqlite3.Cursor object at 0x1006a22d0>
```

Python's triple quotes are handy when creating long strings such as SQL queries.

Now, add some animals to the zoo:

```
>>> curs.execute('INSERT INTO zoo VALUES("duck", 5, 0.0)')
<sqlite3.Cursor object at 0x1006a22d0>
>>> curs.execute('INSERT INTO zoo VALUES("bear", 2, 1000.0)')
<sqlite3.Cursor object at 0x1006a22d0>
```

There's a safer way to insert data, using a *placeholder*:

```
>>> ins = 'INSERT INTO zoo (critter, count, damages) VALUES(?, ?, ?)'
>>> curs.execute(ins, ('weasel', 1, 2000.0))
<sqlite3.Cursor object at 0x1006a22d0>
```

This time, we used three question marks in the SQL to indicate that we plan to insert three values, and then pass those three values as a tuple to the `execute()` function. Placeholders handle tedious details such as quoting. They protect you against *SQL injection*, a kind of external attack that inserts malicious SQL commands into the system (and which is common on the web).

Now, let's see if we can get all our animals out again:

```
>>> curs.execute('SELECT * FROM zoo')
<sqlite3.Cursor object at 0x1006a22d0>
>>> rows = curs.fetchall()
>>> print(rows)
[('duck', 5, 0.0), ('bear', 2, 1000.0), ('weasel', 1, 2000.0)]
```

Let's get them again, but ordered by their counts:

```
>>> curs.execute('SELECT * from zoo ORDER BY count')
<sqlite3.Cursor object at 0x1006a22d0>
>>> curs.fetchall()
[('weasel', 1, 2000.0), ('bear', 2, 1000.0), ('duck', 5, 0.0)]
```

Hey, we wanted them in descending order:

```
>>> curs.execute('SELECT * from zoo ORDER BY count DESC')
<sqlite3.Cursor object at 0x1006a22d0>
>>> curs.fetchall()
[('duck', 5, 0.0), ('bear', 2, 1000.0), ('weasel', 1, 2000.0)]
```

Which type of animal is costing us the most?

```
>>> curs.execute('''SELECT * FROM zoo WHERE
...     damages = (SELECT MAX(damages) FROM zoo)''')
<sqlite3.Cursor object at 0x1006a22d0>
>>> curs.fetchall()
[('weasel', 1, 2000.0)]
```

You would have thought it was the bears. It's always best to check the actual data.

Before we leave SQLite, we need to clean up. If we opened a connection and a cursor, we need to close them when we're done:

```
>>> curs.close()
>>> conn.close()
```

MySQL

MySQL is a very popular open source relational database. Unlike SQLite, it's an actual server, so clients can access it from different devices across the network.

Table 16-3 lists the drivers you can use to access MySQL from Python. For more details on all Python MySQL drivers, see the *python.org* wiki.

Table 16-3. MySQL drivers

Name	Link	Pypi package	Import as	Notes
mysql-client	*https://https://mysqlclient.readthedocs.io*	mysql-connector-python	`MySQLdb`	
MySQL Connector	*http://bit.ly/mysql-cpdg*	mysql-connector-python	`mysql.connector`	
PYMySQL	*https://github.com/petehunt/PyMySQL*	pymysql	`pymysql`	
oursql	*http://pythonhosted.org/oursql*	oursql	`oursql`	Requires the MySQL C client libraries

PostgreSQL

PostgreSQL is a full-featured open source relational database. Indeed in many ways, it's more advanced than MySQL. Table 16-4 presents the Python drivers

you can use to access it.

Table 16-4. PostgreSQL drivers

Name	Link	Pypi package	Import as	Notes
psycopg2	*http://initd.org/psycopg*	psycopg2	psycopg2	Needs `pg_config` from PostgreSQL client tools
py-postgresql	*https://pypi.org/project/py-postgresql*	py-postgresql	postgresql	

The most popular driver is `psycopg2`, but its installation requires the PostgreSQL client libraries.

SQLAlchemy

SQL is not quite the same for all relational databases, and DB-API takes you only so far. Each database implements a particular *dialect* reflecting its features and philosophy. Many libraries try to bridge these differences in one way or another. The most popular cross-database Python library is SQLAlchemy.

It isn't in the standard library, but it's well known and used by many people. You can install it on your system by using this command:

```
$ pip install sqlalchemy
```

You can use SQLAlchemy on several levels:

- The lowest level manages database connection *pools*, executes SQL commands, and returns results. This is closest to the DB-API.

- Next up is the *SQL expression language*, which lets you express queries in a more Python-oriented way.
- Highest is the ORM (Object Relational Model) layer, which uses the SQL Expression Language and binds application code with relational data structures.

As we go along, you'll understand what the terms mean in those levels. SQLAlchemy works with the database drivers documented in the previous sections. You don't need to import the driver; the initial connection string you provide to SQLAlchemy will determine it. That string looks like this:

```
dialect + driver :// user : password @ host : port / dbname
```

The values you put in this string are as follows:

`dialect`
: The database type

`driver`
: The particular driver you want to use for that database

`user` *and* `password`
: Your database authentication strings

`host` *and* `port`
: The database server's location (`: port` is needed only if it's not the standard one for this server)

`dbname`
: The database to initially connect to on the server

Table 16-5 lists the dialects and drivers.

Table 16-5. SQLAlchemy connection

dialect	driver
`sqlite`	`pysqlite` (or omit)
`mysql`	`mysqlconnector`
`mysql`	`pymysql`
`mysql`	`oursql`
`postgresql`	`psycopg2`
`postgresql`	`pypostgresql`

See also the SQLAlchemy details on dialects for MySQL, SQLite, PostgreSQL, and other databases.

The engine layer

First, let's try the lowest level of SQLAlchemy, which does little more than the base DB-API functions.

Let's try it with SQLite, which is already built in to Python. The connection string for SQLite skips the *`host`*, *`port`*, *`user`*, and *`password`*. The *`dbname`* tells SQLite what file to use to store your database. If you omit the *`dbname`*, SQLite builds a database in memory. If the *`dbname`* starts with a slash (/), it's an absolute filename on your computer (as in Linux and macOS; for example, `C:\` on Windows). Otherwise, it's relative to your current directory.

The following segments are all part of one program, separated here for explanation.

To begin, you need to import what we need. The following is an example of an *import alias*, which lets us use the string `sa` to refer to SQLAlchemy methods. I do this mainly because `sa` is a lot easier to type than `sqlalchemy`:

```
>>> import sqlalchemy as sa
```

Connect to the database and create the storage for it in memory (the argument string `'sqlite:///:memory:'` also works):

```
>>> conn = sa.create_engine('sqlite://')
```

Create a database table called `zoo` that comprises three columns:

```
>>> conn.execute('''CREATE TABLE zoo
...     (critter VARCHAR(20) PRIMARY KEY,
...      count INT,
...      damages FLOAT)''')
<sqlalchemy.engine.result.ResultProxy object at 0x1017efb10>
```

Running `conn.execute()` returns a SQLAlchemy object called a `ResultProxy`. You'll soon see what to do with it.

By the way, if you've never made a database table before, congratulations. Check that one off your bucket list.

Now, insert three sets of data into your new empty table:

```
>>> ins = 'INSERT INTO zoo (critter, count, damages) VALUES (?, ?, ?)'
>>> conn.execute(ins, 'duck', 10, 0.0)
<sqlalchemy.engine.result.ResultProxy object at 0x1017efb50>
>>> conn.execute(ins, 'bear', 2, 1000.0)
<sqlalchemy.engine.result.ResultProxy object at 0x1017ef090>
>>> conn.execute(ins, 'weasel', 1, 2000.0)
<sqlalchemy.engine.result.ResultProxy object at 0x1017ef450>
```

Next, ask the database for everything that we just put in:

```
>>> rows = conn.execute('SELECT * FROM zoo')
```

In SQLAlchemy, `rows` is not a list; it's that special `ResultProxy` thing that we can't print directly:

```
>>> print(rows)
<sqlalchemy.engine.result.ResultProxy object at 0x1017ef9d0>
```

However, you can iterate over it like a list, so we can get a row at a time:

```
>>> for row in rows:
...     print(row)
...
('duck', 10, 0.0)
('bear', 2, 1000.0)
('weasel', 1, 2000.0)
```

That was almost the same as the SQLite DB-API example that you saw earlier. The one advantage is that we didn't need to import the database driver at the top; SQLAlchemy figured that out from the connection string. Just changing the connection string would make this code portable to another type of database. Another plus is SQLAlchemy's *connection pooling*, which you can read about at its documentation site.

The SQL Expression Language

The next level up is SQLAlchemy's SQL Expression Language. It introduces functions to create the SQL for various operations. The Expression Language handles more of the SQL dialect differences than the lower-level engine layer does. It can be a handy middle-ground approach for relational database applications.

Here's how to create and populate the `zoo` table. Again, these are successive fragments of a single program.

The import and connection are the same as before:

```
>>> import sqlalchemy as sa
>>> conn = sa.create_engine('sqlite://')
```

To define the `zoo` table, we begin using some of the Expression Language instead of SQL:

```
>>> meta = sa.MetaData()
>>> zoo = sa.Table('zoo', meta,
...     sa.Column('critter', sa.String, primary_key=True),
...     sa.Column('count', sa.Integer),
...     sa.Column('damages', sa.Float)
...    )
>>> meta.create_all(conn)
```

Check out the parentheses in that multiline call in the preceding example. The structure of the `Table()` method matches the structure of the table. Just as our table contains three columns, there are three calls to `Column()` inside the parentheses of the `Table()` method call.

Meanwhile, `zoo` is some magic object that bridges the SQL database world and the Python data structure world.

Insert the data with more Expression Language functions:

```
... conn.execute(zoo.insert(('bear', 2, 1000.0)))
<sqlalchemy.engine.result.ResultProxy object at 0x1017ea910>
>>> conn.execute(zoo.insert(('weasel', 1, 2000.0)))
<sqlalchemy.engine.result.ResultProxy object at 0x1017eab10>
>>> conn.execute(zoo.insert(('duck', 10, 0)))
<sqlalchemy.engine.result.ResultProxy object at 0x1017eac50>
```

Next, create the SELECT statement (`zoo.select()` selects everything from the table represented by the `zoo` object, such as `SELECT * FROM zoo` would do in plain SQL):

```
>>> result = conn.execute(zoo.select())
```

Finally, get the results:

```
>>> rows = result.fetchall()
>>> print(rows)
[('bear', 2, 1000.0), ('weasel', 1, 2000.0), ('duck', 10, 0.0)]
```

The Object-Relational Mapper (ORM)

In the previous section, the `zoo` object was a mid-level connection between SQL and Python. At the top layer of SQLAlchemy, the Object-Relational Mapper (ORM) uses the SQL Expression Language but tries to make the actual database mechanisms invisible. You define classes, and the ORM handles how to get their data in and out of the database. The basic idea behind that complicated phrase, "object-relational mapper," is that you can refer to objects in your code and thus stay close to the way Python likes to operate while still using a relational database.

We'll define a `Zoo` class and hook it into the ORM. This time, we make SQLite use the file *zoo.db* so that we can confirm that the ORM worked.

As in the previous two sections, the snippets that follow are actually one program separated by explanations. Don't worry if you don't understand some if it. The SQLAlchemy documentation has all the details—and this stuff can get complex. I just want you to get an idea of how much work it is to do this, so that you can decide which of the approaches discussed in this chapter suits you.

The initial import is the same, but this time we need another something also:

```
>>> import sqlalchemy as sa
>>> from sqlalchemy.ext.declarative import declarative_base
```

Here, we make the connection:

```
>>> conn = sa.create_engine('sqlite:///zoo.db')
```

Now, we get into SQLAlchemy's ORM. We define the `Zoo` class and associate its attributes with table columns:

```
>>> Base = declarative_base()
>>> class Zoo(Base):
...     __tablename__ = 'zoo'
...     critter = sa.Column('critter', sa.String, primary_key=True)
...     count = sa.Column('count', sa.Integer)
...     damages = sa.Column('damages', sa.Float)
...     def __init__(self, critter, count, damages):
...         self.critter = critter
...         self.count = count
...         self.damages = damages
...     def __repr__(self):
...         return "<Zoo({}, {}, {})>".format(self.critter, self.count,
...           self.damages)
```

The following line magically creates the database and table:

```
>>> Base.metadata.create_all(conn)
```

You can then insert data by creating Python objects. The ORM manages these internally:

```
>>> first = Zoo('duck', 10, 0.0)
>>> second = Zoo('bear', 2, 1000.0)
>>> third = Zoo('weasel', 1, 2000.0)
>>> first
<Zoo(duck, 10, 0.0)>
```

Next, we get the ORM to take us to SQL land. We create a session to talk to the database:

```
>>> from sqlalchemy.orm import sessionmaker
>>> Session = sessionmaker(bind=conn)
>>> session = Session()
```

Within the session, we write the three objects that we created to the database. The `add()` function adds one object, and `add_all()` adds a list:

```
>>> session.add(first)
>>> session.add_all([second, third])
```

Finally, we need to force everything to complete:

```
>>> session.commit()
```

Did it work? Well, it created a *zoo.db* file in the current directory. You can use the command-line `sqlite3` program to check it:

```
$ sqlite3 zoo.db
SQLite version 3.6.12
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .tables
zoo
sqlite> select * from zoo;
duck|10|0.0
bear|2|1000.0
weasel|1|2000.0
```

The purpose of this section was to show what an ORM is and how it works at a high level. The author of SQLAlchemy has written a full tutorial. After reading this, decide which of the following levels would best fit your needs:

- Plain DB-API, as in the earlier SQLite section
- The SQLAlchemy engine
- The SQLAlchemy Expression Language
- The SQLAlchemy ORM

It seems like a natural choice to use an ORM to avoid the complexities of SQL. Should you use one? Some people think ORMs should be avoided, but others think the criticism is overdone. Whoever's right, an ORM is an abstraction, and all abstractions are leaky and break down at some point. When your ORM doesn't do what you want, you must figure out both how it works and how to fix it in SQL. To borrow an internet meme:

> *Some people, when confronted with a problem, think, "I know, I'll use an ORM." Now they have two problems.*

Use ORMs for simple applications or applications that map data pretty directly to database tables. If the application is that simple, you may consider using straight SQL or the SQL Expression Language.

Other Database Access Packages

If you're looking for Python tools that will handle multiple databases, with more features than the bare db-api but less than SQLAlchemy, these are worth a look:

- `dataset` claims the goal "databases for lazy people". It's built on SQLAlchemy and provides a simple ORM for SQL, JSON, and CSV storage.
- `records` bills itself as "SQL for Humans." It supports only SQL queries, using SQLAlchemy internally to handle SQL dialect issues, connection pooling, and other details. Its integration with `tablib` (mentioned at "Tablib") lets you export data to CSV, JSON, and other formats.

NoSQL Data Stores

Relational tables are rectangular, but data come in many shapes and may be very difficult to fit without significant effort and contortion. It's a square peg/round hole problem.

Some nonrelational databases have been written to allow more flexible data definitions as well as to process very large data sets or support custom data operations. They've been collectively labeled *NoSQL* (formerly meaning *no SQL*, now the less confrontational *not only SQL*).

The simplest type of NoSQL databases are *key-value stores*. One popularity ranking shows some that I cover in the following sections.

The dbm Family

The `dbm` formats were around long before the *NoSQL* label was coined. They're simple key-value stores, often embedded in applications such as web browsers to maintain various settings. A dbm database is like a Python dictionary in the following ways:

- You can assign a value to a key, and it's automatically saved to the database on disk.
- You can query a key for its value.

The following is a quick example. The second argument to the following `open()` method is `'r'` to read, `'w'` to write, and `'c'` for both, creating the file if it doesn't exist:

```
>>> import dbm
>>> db = dbm.open('definitions', 'c')
```

To create key-value pairs, just assign a value to a key just as you would a dictionary:

```
>>> db['mustard'] = 'yellow'
>>> db['ketchup'] = 'red'
>>> db['pesto'] = 'green'
```

Let's pause and check what we have so far:

```
>>> len(db)
3
>>> db['pesto']
b'green'
```

Now close, then reopen to see whether it actually saved what we gave it:

```
>>> db.close()
>>> db = dbm.open('definitions', 'r')
```

```
>>> db['mustard']
b'yellow'
```

Keys and values are stored as `bytes`. You cannot iterate over the database object `db`, but you can get the number of keys by using `len()`. `get()` and `setdefault()` work as they do for dictionaries.

Memcached

`memcached` is a fast in-memory key-value *cache* server. It's often put in front of a database, or used to store web server session data.

You can download versions for Linux and macOS as well as Windows. If you want to try out this section, you'll need a running memcached server and Python driver.

There are many Python drivers; one that works with Python 3 is `python3-memcached`, which you can install by using this command:

```
$ pip install python-memcached
```

To use it, connect to a memcached server, after which you can do the following:

- Set and get values for keys
- Increment or decrement a value
- Delete a key

Data keys and values are *not* persistent, and data that you wrote earlier might disappear. This is inherent in memcached—it's a cache server, not a database, and it avoids running out of memory by discarding old data.

You can connect to multiple memcached servers at the same time. In this next example, we're just talking to one on the same computer:

```
>>> import memcache
>>> db = memcache.Client(['127.0.0.1:11211'])
```

```
>>> db.set('marco', 'polo')
True
>>> db.get('marco')
'polo'
>>> db.set('ducks', 0)
True
>>> db.get('ducks')
0
>>> db.incr('ducks', 2)
2
>>> db.get('ducks')
2
```

Redis

Redis is a *data structure server*. It handles keys and their values, but the values are richer than those in other key-value stores. Like memcached, all of the data in a Redis server should fit in memory. Unlike memcached, Redis can do the following:

- Save data to disk for reliability and restarts
- Keep old data
- Provide more data structures than simple strings

The Redis data types are a close match to Python's, and a Redis server can be a useful intermediary for one or more Python applications to share data. I've found it so useful that it's worth a little extra coverage here.

The Python driver `redis-py` has its source code and tests on GitHub as well as documentation. You can install it by using this command:

```
$ pip install redis
```

The Redis server has good documentation. If you install and start the Redis server on your local computer (with the network nickname `localhost`), you can try the programs in the following sections.

Strings

A key with a single value is a Redis *string*. Simple Python data types are automatically converted. Connect to a Redis server at some host (default is `localhost`) and port (default is `6379`):

```
>>> import redis
>>> conn = redis.Redis()
```

Connecting to `redis.Redis('localhost')` or `redis.Redis('localhost', 6379)` would have given the same result.

List all keys (none so far):

```
>>> conn.keys('*')
[]
```

Set a simple string (key `'secret'`), integer (key `'carats'`), and float (key `'fever'`):

```
>>> conn.set('secret', 'ni!')
True
>>> conn.set('carats', 24)
True
>>> conn.set('fever', '101.5')
True
```

Get the values back (as Python `byte` values) by key:

```
>>> conn.get('secret')
b'ni!'
>>> conn.get('carats')
b'24'
>>> conn.get('fever')
b'101.5'
```

Here, the `setnx()` method sets a value only if the key does not exist:

```
>>> conn.setnx('secret', 'icky-icky-icky-ptang-zoop-boing!')
False
```

It failed because we had already defined `'secret'`:

```
>>> conn.get('secret')
b'ni!'
```

The `getset()` method returns the old value and sets it to a new one at the same time:

```
>>> conn.getset('secret', 'icky-icky-icky-ptang-zoop-boing!')
b'ni!'
```

Let's not get too far ahead of ourselves. Did it work?

```
>>> conn.get('secret')
b'icky-icky-icky-ptang-zoop-boing!'
```

Now, get a substring by using `getrange()` (as in Python, offset `0` means start, and `-1` means end):

```
>>> conn.getrange('secret', -6, -1)
b'boing!'
```

Replace a substring by using `setrange()` (using a zero-based offset):

```
>>> conn.setrange('secret', 0, 'ICKY')
32
>>> conn.get('secret')
b'ICKY-icky-icky-ptang-zoop-boing!'
```

Next, set multiple keys at once by using `mset()`:

```
>>> conn.mset({'pie': 'cherry', 'cordial': 'sherry'})
True
```

Get more than one value at once by using `mget()`:

```
>>> conn.mget(['fever', 'carats'])
[b'101.5', b'24']
```

Delete a key by using `delete()`:

```
>>> conn.delete('fever')
True
```

Increment by using the `incr()` or `incrbyfloat()` commands, and decrement with `decr()`:

```
>>> conn.incr('carats')
25
>>> conn.incr('carats', 10)
35
>>> conn.decr('carats')
34
>>> conn.decr('carats', 15)
19
>>> conn.set('fever', '101.5')
True
>>> conn.incrbyfloat('fever')
102.5
>>> conn.incrbyfloat('fever', 0.5)
103.0
```

There's no `decrbyfloat()`. Use a negative increment to reduce the fever:

```
>>> conn.incrbyfloat('fever', -2.0)
101.0
```

Lists

Redis lists can contain only strings. The list is created when you do your first insertion. Insert at the beginning by using `lpush()`:

```
>>> conn.lpush('zoo', 'bear')
1
```

Insert more than one item at the beginning:

```
>>> conn.lpush('zoo', 'alligator', 'duck')
3
```

Insert before or after a value by using `linsert()`:

```
>>> conn.linsert('zoo', 'before', 'bear', 'beaver')
4
>>> conn.linsert('zoo', 'after', 'bear', 'cassowary')
5
```

Insert at an offset by using `lset()` (the list must exist already):

```
>>> conn.lset('zoo', 2, 'marmoset')
True
```

Insert at the end by using `rpush()`:

```
>>> conn.rpush('zoo', 'yak')
6
```

Get the value at an offset by using `lindex()`:

```
>>> conn.lindex('zoo', 3)
b'bear'
```

Get the values in an offset range by using `lrange()` (0 to -1 for all):

```
>>> conn.lrange('zoo', 0, 2)
[b'duck', b'alligator', b'marmoset']
```

Trim the list with `ltrim()`, keeping only those in a range of offsets:

```
>>> conn.ltrim('zoo', 1, 4)
True
```

Get a range of values (use `0` to `-1` for all) by using `lrange()`:

```
>>> conn.lrange('zoo', 0, -1)
[b'alligator', b'marmoset', b'bear', b'cassowary']
```

Chapter 15 shows you how you can use Redis lists and *publish-subscribe* to implement job queues.

Hashes

Redis *hashes* are similar to Python dictionaries but can contain only strings. Also, you can go only one level deep, not make deep-nested structures. Here are examples that create and play with a Redis hash called `song`:

Set the fields `do` and `re` in hash `song` at once by using `hmset()`:

```
>>> conn.hmset('song', {'do': 'a deer', 're': 'about a deer'})
True
```

Set a single field value in a hash by using `hset()`:

```
>>> conn.hset('song', 'mi', 'a note to follow re')
1
```

Get one field's value by using `hget()`:

```
>>> conn.hget('song', 'mi')
b'a note to follow re'
```

Get multiple field values by using `hmget()`:

```
>>> conn.hmget('song', 're', 'do')
[b'about a deer', b'a deer']
```

Get all field keys for the hash by using `hkeys()`:

```
>>> conn.hkeys('song')
[b'do', b're', b'mi']
```

Get all field values for the hash by using `hvals()`:

```
>>> conn.hvals('song')
[b'a deer', b'about a deer', b'a note to follow re']
```

Get the number of fields in the hash by using `hlen()`:

```
>>> conn.hlen('song')
3
```

Get all field keys and values in the hash by using `hgetall()`:

```
>>> conn.hgetall('song')
{b'do': b'a deer', b're': b'about a deer', b'mi': b'a note to follow re'}
```

Set a field if its key doesn't exist by using `hsetnx()`:

```
>>> conn.hsetnx('song', 'fa', 'a note that rhymes with la')
1
```

Sets

Redis sets are similar to Python sets, as you'll see in the following examples.

Add one or more values to a set:

```
>>> conn.sadd('zoo', 'duck', 'goat', 'turkey')
3
```

Get the number of values from the set:

```
>>> conn.scard('zoo')
3
```

Get all of the set's values:

```
>>> conn.smembers('zoo')
{b'duck', b'goat', b'turkey'}
```

Remove a value from the set:

```
>>> conn.srem('zoo', 'turkey')
True
```

Let's make a second set to show some set operations:

```
>>> conn.sadd('better_zoo', 'tiger', 'wolf', 'duck')
0
```

Intersect (get the common members of) the `zoo` and `better_zoo` sets:

```
>>> conn.sinter('zoo', 'better_zoo')
{b'duck'}
```

Get the intersection of `zoo` and `better_zoo`, and store the result in the set `fowl_zoo`:

```
>>> conn.sinterstore('fowl_zoo', 'zoo', 'better_zoo')
1
```

Who's in there?

```
>>> conn.smembers('fowl_zoo')
{b'duck'}
```

Get the union (all members) of `zoo` and `better_zoo`:

```
>>> conn.sunion('zoo', 'better_zoo')
{b'duck', b'goat', b'wolf', b'tiger'}
```

Store that union result in the set `fabulous_zoo`:

```
>>> conn.sunionstore('fabulous_zoo', 'zoo', 'better_zoo')
4
>>> conn.smembers('fabulous_zoo')
{b'duck', b'goat', b'wolf', b'tiger'}
```

What does `zoo` have that `better_zoo` doesn't? Use `sdiff()` to get the set difference, and `sdiffstore()` to save it in the `zoo_sale` set:

```
>>> conn.sdiff('zoo', 'better_zoo')
{b'goat'}
>>> conn.sdiffstore('zoo_sale', 'zoo', 'better_zoo')
1
>>> conn.smembers('zoo_sale')
{b'goat'}
```

Sorted sets

One of the most versatile Redis data types is the *sorted set*, or *zset*. It's a set of unique values, but each value has an associated floating-point *score*. You can access each item by its value or score. Sorted sets have many uses:

- Leader boards
- Secondary indexes
- Timeseries, using timestamps as scores

We show the last use case, tracking user logins via timestamps. We're using the Unix *epoch* value (more on this in Chapter 15) that's returned by the Python `time()` function:

```
>>> import time
>>> now = time.time()
>>> now
1361857057.576483
```

Let's add our first guest, looking nervous:

```
>>> conn.zadd('logins', 'smeagol', now)
1
```

Five minutes later, another guest:

```
>>> conn.zadd('logins', 'sauron', now+(5*60))
1
```

Two hours later:

```
>>> conn.zadd('logins', 'bilbo', now+(2*60*60))
1
```

One day later, not hasty:

```
>>> conn.zadd('logins', 'treebeard', now+(24*60*60))
1
```

In what order did `bilbo` arrive?

```
>>> conn.zrank('logins', 'bilbo')
2
```

When was that?

```
>>> conn.zscore('logins', 'bilbo')
1361864257.576483
```

Let's see everyone in login order:

```
>>> conn.zrange('logins', 0, -1)
[b'smeagol', b'sauron', b'bilbo', b'treebeard']
```

With their times, please:

```
>>> conn.zrange('logins', 0, -1, withscores=True)
[(b'smeagol', 1361857057.576483), (b'sauron', 1361857357.576483),
(b'bilbo', 1361864257.576483), (b'treebeard', 1361943457.576483)]
```

Caches and expiration

All Redis keys have a time-to-live, or *expiration date*. By default, this is forever. We can use the `expire()` function to instruct Redis how long to keep the key. The value is a number of seconds:

```
>>> import time
>>> key = 'now you see it'
>>> conn.set(key, 'but not for long')
True
>>> conn.expire(key, 5)
True
```

```
>>> conn.ttl(key)
5
>>> conn.get(key)
b'but not for long'
>>> time.sleep(6)
>>> conn.get(key)
>>>
```

The `expireat()` command expires a key at a given epoch time. Key expiration is useful to keep caches fresh and to limit login sessions. An analogy: in the refrigerated room behind the milk racks at your grocery store, store employees yank those gallons as they reach their freshness dates.

Document Databases

A *document database* is a NoSQL database that stores data with varying fields. Compared to a relational table (rectangular, with the same columns in every row) such data is "ragged," with varying fields (columns) per row, and even nested fields. You could handle data like this in memory with Python dictionaries and lists, or store it as JSON files. To store such data in a relational database table, you would need to define every possible column and use nulls for missing data.

ODM can stand for Object Data Manager or Object Document Mapper (at least they agree on the *O* part). An ODM is the document database counterpart of a relational database ORM. Some popular document databases and tools (drivers and ODMs) are listed in Table 16-6.

Table 16-6. Document databases

Database	Python API
Mongo	tools
DynamoDB	`boto3`
CouchDB	`couchdb`

NOTE

PostgreSQL can do some of the things that document databases do. Some of its extensions allow it to escape relational orthodoxy while keeping features like transactions, data validation, and foreign keys: 1) multidimensional *arrays*—store more than one value in a table cell; 2) *jsonb*—store JSON data in a cell, with full indexing and querying.

Time Series Databases

Time series data may be collected at fixed intervals (such as computer performance metrics) or at random times, which has led to many storage methods. Among many of these, some with Python support are listed in Table 16-7.

Table 16-7. Temporal databases

Database	Python API
InfluxDB	`influx-client`
kdb+	PyQ
Prometheus	`prometheus_client`
TimescaleDB	(PostgreSQL clients)
OpenTSDB	`potsdb`
PyStore	`PyStore`

Graph Databases

For our last case of data that need its own database category, we have *graphs*: *nodes* (data) connected by *edges* or *vertices* (relationships). An individual Twitter *user* could be a node, with edges to other users like *following* and *followed*.

Graph data has become more visible with the growth of social media, where the value is in the connections as much as the content. Some popular graph databases are outlined in Table 16-8.

Table 16-8. Graph databases

Database	Python API
Neo4J	`py2neo`
OrientDB	`pyorient`
ArangoDB	`pyArango`

Other NoSQL

The NoSQL servers listed here handle data larger than memory, and many of them use multiple computers. Table 16-9 presents notable servers and their Python libraries.

Table 16-9. NoSQL databases

Database	Python API
Cassandra	`pycassa`
CouchDB	`couchdb-python`
HBase	`happybase`
Kyoto Cabinet	`kyotocabinet`
MongoDB	`mongodb`
Pilosa	`python-pilosa`
Riak	`riak-python-client`

Full-Text Databases

Finally, there's a special category of databases for *full-text* search. They index everything, so you can find that poem that talks about windmills and giant wheels of cheese. You can see some popular open source examples along with their Python APIs in Table 16-10.

Table 16-10. Full-text databases

Site	Python API
Lucene	`pylucene`
Solr	`SolPython`
ElasticSearch	`elasticsearch`
Sphinx	`sphinxapi`
Xapian	`xappy`
Whoosh	(written in Python, includes an API)

Coming Up

The previous chapter was about interleaving code in time (*concurrency*). The next one is about moving data through space (*networking*), which can be used for concurrency and other reasons.

Things to Do

16.1 Save the following text lines to a file called *books.csv* (notice that if the fields are separated by commas, you need to surround a field with quotes if it contains a comma):

```
author,book
J R R Tolkien,The Hobbit
Lynne Truss,"Eats, Shoots & Leaves"
```

16.2 Use the `csv` module and its `DictReader` method to read *books.csv* to the variable `books`. Print the values in `books`. Did `DictReader` handle the quotes and commas in the second book's title?

16.3 Create a CSV file called *books2.csv* by using these lines:

```
title,author,year
The Weirdstone of Brisingamen,Alan Garner,1960
Perdido Street Station,China Miéville,2000
Thud!,Terry Pratchett,2005
The Spellman Files,Lisa Lutz,2007
Small Gods,Terry Pratchett,1992
```

16.4 Use the `sqlite3` module to create a SQLite database called *books.db* and a table called `books` with these fields: `title` (text), `author` (text), and `year` (integer).

16.5 Read *books2.csv* and insert its data into the `book` table.

16.6 Select and print the `title` column from the `book` table in alphabetical order.

16.7 Select and print all columns from the `book` table in order of publication.

16.8 Use the `sqlalchemy` module to connect to the sqlite3 database *books.db* that you just made in exercise 16.4. As in 16.6, select and print the `title` column from the `book` table in alphabetical order.

16.9 Install the Redis server and the Python `redis` library (`pip install redis`) on your computer. Create a Redis hash called `test` with the fields `count` (`1`) and `name` (`'Fester Bestertester'`). Print all the fields for `test`.

16.10 Increment the `count` field of `test` and print it.

1 Alas, not XML yet.