CHAPTER 8

Data Has to Go Somewhere

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

An active program accesses data that is 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 tradeoffs 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 "Files" on page 247.

This is also the first chapter to show examples of nonstandard Python modules; that is, Python code apart from the standard library. You'll install them by using the `pip` command, which is painless. There are more details on its usage in Appendix D.

File Input/Output

The simplest kind of persistence is a plain old file, sometimes called a *flat file*. This is just a sequence of bytes stored under a *filename*. You *read* from a file into memory and *write* from memory to a file. Python makes these jobs easy. Its file operations were modeled on the familiar and popular Unix equivalents.

Before reading or writing a file, you need to *open* it:

```
fileobj = open( filename, mode )
```

Here's a brief explanation of the pieces of this call:

- *`fileobj`* is the file object returned by `open()`
- *`filename`* is the string name of the file
- *`mode`* is a string indicating the file's type and what you want to do with it

The first letter of *`mode`* indicates the *operation*:

- `r` means read.
- `w` means write. If the file doesn't exist, it's created. If the file does exist, it's overwritten.
- `x` means write, but only if the file does *not* already exist.
- `a` means append (write after the end) if the file exists.

The second letter of *`mode`* is the file's *type*:

- `t` (or nothing) means text.
- `b` means binary.

After opening the file, you call functions to read or write data; these will be shown in the examples that follow.

Last, you need to *close* the file.

Let's create a file from a Python string in one program and then read it back in the next.

Write a Text File with write()

For some reason, there aren't many limericks about special relativity. This one will just have to do for our data source:

```
>>> poem = '''There was a young lady named Bright,
... Whose speed was far faster than light;
... She started one day
... In a relative way,
... And returned on the previous night.'''
>>> len(poem)
150
```

The following code writes the entire poem to the file `'relativity'` in one call:

```
>>> fout = open('relativity', 'wt')
>>> fout.write(poem)
```

```
150
>>> fout.close()
```

The `write()` function returns the number of bytes written. It does not add any spaces or newlines, as `print()` does. You can also `print()` to a text file:

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout)
>>> fout.close()
```

This brings up the question: should I use `write()` or `print()`? By default, `print()` adds a space after each argument and a newline at the end. In the previous example, it appended a newline to the `relativity` file. To make `print()` work like `write()`, pass the following two arguments:

- `sep` (separator, which defaults to a space, `' '`)
- `end` (end string, which defaults to a newline, `'\n'`)

`print()` uses the defaults unless you pass something else. We'll pass empty strings to suppress all of the fussiness normally added by `print()`:

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout, sep='', end='')
>>> fout.close()
```

If you have a large source string, you can also write chunks until the source is done:

```
>>> fout = open('relativity', 'wt')
>>> size = len(poem)
>>> offset = 0
>>> chunk = 100
>>> while True:
...     if offset > size:
...          break
...     fout.write(poem[offset:offset+chunk])
...     offset += chunk
...
100
50
>>> fout.close()
```

This wrote 100 characters on the first try and the last 50 characters on the next.

If the `relativity` file is precious to us, let's see if using mode `x` really protects us from overwriting it:

```
>>> fout = open('relativity', 'xt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'relativity'
```

You can use this with an exception handler:

```
>>> try:
...     fout = open('relativity', 'xt')]
...     fout.write('stomp stomp stomp')
... except FileExistsError:
...     print('relativity already exists!. That was a close one.')
...
relativity already exists!. That was a close one.
```

Read a Text File with read(), readline(), or readlines()

You can call `read()` with no arguments to slurp up the entire file at once, as shown in the example that follows. Be careful when doing this with large files; a gigabyte file will consume a gigabyte of memory.

```
>>> fin = open('relativity', 'rt' )
>>> poem = fin.read()
>>> fin.close()
>>> len(poem)
150
```

You can provide a maximum character count to limit how much `read()` returns at one time. Let's read 100 characters at a time and append each chunk to a `poem` string to rebuild the original:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> chunk = 100
>>> while True:
...     fragment = fin.read(chunk)
...     if not fragment:
...         break
...     poem += fragment
...
>>> fin.close()
>>> len(poem)
150
```

After you've read all the way to the end, further calls to `read()` will return an empty string (`''`), which is treated as `False` in `if not fragment`. This breaks out of the `while True` loop.

You can also read the file a line at a time by using `readline()`. In this next example, we'll append each line to the `poem` string to rebuild the original:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> while True:
...     line = fin.readline()
...     if not line:
...         break
...     poem += line
...
```

```
>>> fin.close()
>>> len(poem)
150
```

For a text file, even a blank line has a length of one (the newline character), and is evaluated as `True`. When the file has been read, `readline()` (like `read()`) also returns an empty string, which is also evaluated as `False`.

The easiest way to read a text file is by using an *iterator*. This returns one line at a time. It's similar to the previous example, but with less code:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> for line in fin:
...     poem += line
...
>>> fin.close()
>>> len(poem)
150
```

All of the preceding examples eventually built the single string `poem`. The `read lines()` call reads a line at a time, and returns a list of one-line strings:

```
>>> fin = open('relativity', 'rt' )
>>> lines = fin.readlines()
>>> fin.close()
>>> print(len(lines), 'lines read')
5 lines read
>>> for line in lines:
...     print(line, end='')
...
There was a young lady named Bright,
Whose speed was far faster than light;
She started one day
In a relative way,
And returned on the previous night.>>>
```

We told `print()` to suppress the automatic newlines because the first four lines already had them. The last line did not, causing the interactive prompt >>> to occur right after the last line.

Write a Binary File with write()

If you include a `'b'` in the *mode* string, the file is opened in binary mode. In this case, you read and write `bytes` instead of a string.

We don't have a binary poem lying around, so we'll just generate the 256 byte values from 0 to 255:

```
>>> bdata = bytes(range(0, 256))
>>> len(bdata)
256
```

Open the file for writing in binary mode and write all the data at once:

```
>>> fout = open('bfile', 'wb')
>>> fout.write(bdata)
256
>>> fout.close()
```

Again, `write()` returns the number of bytes written.

As with text, you can write binary data in chunks:

```
>>> fout = open('bfile', 'wb')
>>> size = len(bdata)
>>> offset = 0
>>> chunk = 100
>>> while True:
...     if offset > size:
...          break
...     fout.write(bdata[offset:offset+chunk])
...     offset += chunk
...
100
100
56
>>> fout.close()
```

Read a Binary File with read()

This one is simple; all you need to do is just open with `'rb'`:

```
>>> fin = open('bfile', 'rb')
>>> bdata = fin.read()
>>> len(bdata)
256
>>> fin.close()
```

Close Files Automatically by Using with

If you forget to close a file that you've opened, it will be closed by Python after it's no longer referenced. This means that if you open a file within a function and don't close it explicitly, it will be closed automatically when the function ends. But you might have opened the file in a long-running function or the main section of the program. The file should be closed to force any remaining writes to be completed.

Python has *context managers* to clean up things such as open files. You use the form `with` *`expression`* `as` *`variable`*:

```
>>> with open('relativity', 'wt') as fout:
...     fout.write(poem)
...
```

That's it. After the block of code under the context manager (in this case, one line) completes (normally *or* by a raised exception), the file is closed automatically.

Change Position with seek()

As you read and write, Python keeps track of where you are in the file. The `tell()` function returns your current offset from the beginning of the file, in bytes. The `seek()` function lets you jump to another byte offset in the file. This means that you don't have to read every byte in a file to read the last one; you can `seek()` to the last one and just read one byte.

For this example, use the 256-byte binary file `'bfile'` that you wrote earlier:

```
>>> fin = open('bfile', 'rb')
>>> fin.tell()
0
```

Use `seek()` to one byte before the end of the file:

```
>>> fin.seek(255)
255
```

Read until the end of the file:

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```

`seek()` also returns the current offset.

You can call `seek()` with a second argument: `seek(` *`offset`*, *`origin`* `)`:

- If `origin` is `0` (the default), go *`offset`* bytes from the start
- If `origin` is `1`, go *`offset`* bytes from the current position
- If `origin` is `2`, go *`offset`* bytes relative to the end

These values are also defined in the standard `os` module:

```
>>> import os
>>> os.SEEK_SET
0
>>> os.SEEK_CUR
1
>>> os.SEEK_END
2
```

So, we could have read the last byte in different ways:

```
>>> fin = open('bfile', 'rb')
```

One byte before the end of the file:

```
>>> fin.seek(-1, 2)
255
>>> fin.tell()
255
```

Read until the end of the file:

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```

You don't need to call `tell()` for `seek()` to work. I just wanted to show that they both report the same offset.

Here's an example of seeking from the current position in the file:

```
>>> fin = open('bfile', 'rb')
```

This next example ends up two bytes before the end of the file:

```
>>> fin.seek(254, 0)
254
>>> fin.tell()
254
```

Now, go forward one byte:

```
>>> fin.seek(1, 1)
255
>>> fin.tell()
255
```

Finally, read until the end of the file:

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```

These functions are most useful for binary files. You can use them with text files, but unless the file is ASCII (one byte per character), you would have a hard time calculating offsets. These would depend on the text encoding, and the most popular encoding (UTF-8) uses varying numbers of bytes per character.

Structured 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 depending on the source you use means "YAML Ain't Markup Language;" you'll need to research that one yourself).
- 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.

- 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'`.
- There can be column names in the first line.

First, we'll 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'll 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]  # This uses a list comprehension
...
>>> print(villains)
[['Doctor', 'No'], ['Rosa', 'Klebb'], ['Mister', 'Big'],
['Auric', 'Goldfinger'], ['Ernst', 'Blofeld']]
```

Take a moment to think about list comprehensions (feel free to go to "Comprehensions" on page 84 and brush up on that syntax). We took advantage of the structure created by the `reader()` function. It obligingly created rows in the `cin` object that we can 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)
[{'last': 'No', 'first': 'Doctor'},
{'last': 'Klebb', 'first': 'Rosa'},
{'last': 'Big', 'first': 'Mister'},
{'last': 'Goldfinger', 'first': 'Auric'},
{'last': 'Blofeld', 'first': 'Ernst'}]
```

Let's rewrite the CSV file by using the new `DictWriter()` function. We'll 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', 'wt') as fout:
    cout = csv.DictWriter(fout, ['first', 'last'])
    cout.writeheader()
    cout.writerows(villains)
```

That creates a `villains` file with a header line:

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

Now we'll read it back. By omitting the `fieldnames` argument in the `DictReader()` call, we instruct 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', 'rt') as fin:
...     cin = csv.DictReader(fin)
...     villains = [row for row in cin]
...
>>> print(villains)
[{'last': 'No', 'first': 'Doctor'},
{'last': 'Klebb', 'first': 'Rosa'},
{'last': 'Big', 'first': 'Mister'},
{'last': 'Goldfinger', 'first': 'Auric'},
{'last': 'Blofeld', 'first': 'Ernst'}]
```

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 the most prominent *markup* format that suits the bill. 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 `ElementTree`. 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:

`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.

HTML

Enormous amounts of data are saved as Hypertext Markup Language (HTML), the basic document format of the Web. The problem is so much of it doesn't follow the HTML rules, which can make it difficult to parse. Also, much of HTML is intended more to format output than interchange data. Because this chapter is intended to describe fairly well-defined data formats, I have separated out the discussion about HTML to Chapter 9.

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 9.

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. As always with standard dictionaries, the order in which you get the keys varies.

You might get an exception while trying to encode or decode some objects, including objects such as datetime (covered in detail in "Calendars and Clocks" on page 256), as demonstrated here.

```
>>> import datetime
>>> 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 (coming in Chapter 10):

```
>>> 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 "Inheritance" on page 128. Python's JSON documentation gives an example of this for complex numbers, which also makes JSON play dead. Let's modify it for datetime:

```
>>> 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 only need to override its 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 date time.datetime. Because everything in Python is an object, isinstance() works everywhere:

```
>>> 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
```

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().

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. 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'
```

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 war is peace for a description of how unprotected YAML loading compromised the Ruby on Rails platform.

A 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 ten nested entities, each expanding the lower level ten 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;&lol1;&lol1;">
 <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
 <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
 <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
 <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
 <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
 <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
 <!ENTITY lol9 "&lol8;&lol8;&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 frontend for the other libraries:

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

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.

Other Interchange Formats

These binary data interchange formats are usually more compact and faster than XML or JSON:

- MsgPack
- Protocol Buffers
- Avro
- Thrift

Because they are binary, none can be easily edited by a human with a text editor.

Serialize by Using pickle

Saving data structures to a file is called *serializing*. Formats such as JSON might require some custom converters to serialize all the data types from a Python program. Python provides the `pickle` module to save and restore any object in a special binary format.

Remember how JSON lost its mind when encountering a `datetime` object? Not a problem for `pickle`:

```
>>> import pickle
>>> import datetime
>>> now1 = datetime.datetime.utcnow()
>>> pickled = pickle.dumps(now1)
>>> now2 = pickle.loads(pickled)
>>> now1
datetime.datetime(2014, 6, 22, 23, 24, 19, 195722)
>>> now2
datetime.datetime(2014, 6, 22, 23, 24, 19, 195722)
```

`pickle` works with your own classes and objects, too. We'll define a little class called `Tiny` that returns the string `'tiny'` when treated as a string:

```
>>> import pickle
>>> class Tiny():
...     def __str__(self):
...        return 'tiny'
...
>>> obj1 = Tiny()
>>> obj1
<__main__.Tiny object at 0x10076ed10>
>>> str(obj1)
'tiny'
>>> pickled = pickle.dumps(obj1)
>>> pickled
b'\x80\x03c__main__\nTiny\nq\x00)\x81q\x01.'
>>> obj2 = pickle.loads(pickled)
>>> obj2
<__main__.Tiny object at 0x10076e550>
>>> str(obj2)
'tiny'
```

`pickled` is the pickled binary string made from the object `obj1`. We converted that back to the object `obj2` to make a copy of `obj1`. Use `dump()` to pickle to a file, and `load()` to unpickle from one.

Because `pickle` can create Python objects, the same security warnings that were discussed in earlier sections apply. Don't unpickle something that you don't trust.

Structured 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.

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. If you have a binary `xls` file, `xlrd` is a third-party package for reading and writing.

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 tera-