GOVERNMENT ARTS AND SCIENCE COLLEGE, KOMARAPALAYAM

DEPARTMENT OF COMPUTER SCIENCE

COURSE	: M.Sc (COMPUTER SCIENCE)
SEMESTER	: III
SUBJECT NAME	: OPEN SOURCE COMPUTING
SUBJECT CODE	: 19PCS09
HANDLED BY	: Dr.V.KARTHIKEYANI

ASSISTANT PROFESSOR

DEPARTMENT OF COMPUTER SCIENCE

GOVERNMENT ARTS AND SCIENCE COLLEGE

KOMARAPALAYAM – 638 183

Scanned with CamScanner

UNIT-III

Data Types: Text Strings – Binary Data. **Storing and Retrieving Data**: File Input/Output – Structured Text Files – Structured Binary Files - Relational Databases – NoSQL Data Stores.

UNIT-IV

Web: Web Clients – Web Servers – Web Services and Automation – **Systems:** Files – Directories – Programs and Processes – Calendars and Clocks

Scanned with CamScanner

III - Unit

CHAPTER 7

Mangle Data Like a Pro

In this chapter, you'll learn many techniques for taming data. Most of them concern these built-in Python data types:

strings
: Sequences of *Unicode* characters, used for text data.

bytes and bytearrays
: Sequences of eight-bit integers, used for binary data.

Text Strings

Text is the most familiar type of data to most readers, so we'll begin with some of the powerful features of text strings in Python.

Unicode

All of the text examples in this book thus far have been plain old ASCII. ASCII was defined in the 1960s, when computers were the size of refrigerators and only slightly better at performing computations. The basic unit of computer storage is the *byte*, which can store 256 unique values in its eight *bits*. For various reasons, ASCII only used 7 bits (128 unique values): 26 uppercase letters, 26 lowercase letters, 10 digits, some punctuation symbols, some spacing characters, and some nonprinting control codes.

Unfortunately, the world has more letters than ASCII provides. You could have a hot dog at a diner, but never a Gewürztraminer[1] at a café. Many attempts have been made to add more letters and symbols, and you'll see them at times. Just a couple of those include:

1. This wine has an umlaut in Germany, but loses it in France.

Scanned with CamScanner
Scanned with CamScanner

- *Latin-1*, or *ISO 8859-1*
- Windows code page *1252*

Each of these uses all eight bits, but even that's not enough, especially when you need non-European languages. *Unicode* is an ongoing international standard to define the characters of all the world's languages, plus symbols from mathematics and other fields.

> Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language.
>
> — The Unicode Consortium

The Unicode Code Charts page has links to all the currently defined character sets with images. The latest version (6.2) defines over 110,000 characters, each with a unique name and identification number. The characters are divided into eight-bit sets called *planes*. The first 256 planes are the *basic multilingual planes*. See the Wikipedia page about Unicode planes for details.

Python 3 Unicode strings

Python 3 strings are Unicode strings, not byte arrays. This is the single largest change from Python 2, which distinguished between normal byte strings and Unicode character strings.

If you know the Unicode ID or name for a character, you can use it in a Python string. Here are some examples:

- A `\u` followed by *four* hex numbers[2] specifies a character in one of Unicode's 256 basic multilingual planes. The first two are the plane number (`00` to `FF`), and the next two are the index of the character within the plane. Plane `00` is good old ASCII, and the character positions within that plane are the same as ASCII.
- For characters in the higher planes, we need more bits. The Python escape sequence for these is `\U` followed by *eight* hex characters; the leftmost ones need to be `0`.
- For all characters, `\N{ name }` lets you specify it by its standard *name*. The Unicode Character Name Index page lists these.

The Python `unicodedata` module has functions that translate in both directions:

- `lookup()`—Takes a case-insensitive name and returns a Unicode character
- `name()`—Takes a Unicode character and returns an uppercase name

2. Base 16, specified with characters 0-9 and A-F.

Scanned with CamScanner
Scanned with CamScanner

In the following example, we'll write a test function that takes a Python Unicode character, looks up its name, and looks up the character again from the name (it should match the original character):

```
>>> def unicode_test(value):
...     import unicodedata
...     name = unicodedata.name(value)
...     value2 = unicodedata.lookup(name)
...     print('value="%s", name="%s", value2="%s"' % (value, name, value2))
...
```

Let's try some characters, beginning with a plain ASCII letter:

```
>>> unicode_test('A')
value="A", name="LATIN CAPITAL LETTER A", value2="A"
```

ASCII punctuation:

```
>>> unicode_test('$')
value="$", name="DOLLAR SIGN", value2="$"
```

A Unicode currency character:

```
>>> unicode_test('\u00a2')
value="¢", name="CENT SIGN", value2="¢"
```

Another Unicode currency character:

```
>>> unicode_test('\u20ac')
value="€", name="EURO SIGN", value2="€"
```

The only problem you could potentially run into is limitations in the font you're using to display text. All fonts do not have images for all Unicode characters, and might display some placeholder character. For instance, here's the Unicode symbol for `SNOWMAN`, like symbols in dingbat fonts:

```
>>> unicode_test('\u2603')
value="☃", name="SNOWMAN", value2="☃"
```

Suppose that we want to save the word café in a Python string. One way is to copy and paste it from a file or website and hope that it works:

```
>>> place = 'café'
>>> place
'café'
```

This worked because I copied and pasted from a source that used UTF-8 encoding (which you'll see in a few pages) for its text.

How can we specify that final é character? If you look at character index for E, you see that the name `E WITH ACUTE, LATIN SMALL LETTER` has the value `00E9`. Let's check with the `name()` and `lookup()` functions that we were just playing with. First give the code to get the name:

Scanned with CamScanner
Scanned with CamScanner

```
>>> unicodedata.name('\u00e9')
'LATIN SMALL LETTER E WITH ACUTE'
```

Next, give the name to look up the code:

```
>>> unicodedata.lookup('E WITH ACUTE, LATIN SMALL LETTER')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: "undefined character name 'E WITH ACUTE, LATIN SMALL LETTER'"
```

> The names listed on the Unicode Character Name Index page were reformatted to make them sort nicely for display. To convert them to their real Unicode names (the ones that Python uses), remove the comma and move the part of the name that was after the comma to the beginning. Accordingly, change `E WITH ACUTE, LATIN SMALL LETTER` to `LATIN SMALL LETTER E WITH ACUTE`:
>
> ```
> >>> unicodedata.lookup('LATIN SMALL LETTER E WITH ACUTE')
> 'é'
> ```

Now, we can specify the string `café` by code or by name:

```
>>> place = 'caf\u00e9'
>>> place
'café'
>>> place = 'caf\N{LATIN SMALL LETTER E WITH ACUTE}'
>>> place
'café'
```

In the preceding snippet, we inserted the é directly in the string, but we can also build a string by appending:

```
>>> u_umlaut = '\N{LATIN SMALL LETTER U WITH DIAERESIS}'
>>> u_umlaut
'ü'
>>> drink = 'Gew' + u_umlaut + 'rztraminer'
>>> print('Now I can finally have my', drink, 'in a', place)
Now I can finally have my Gewürztraminer in a café
```

The string `len` function counts Unicode *characters*, not bytes:

```
>>> len('$')
1
>>> len('\U0001f47b')
1
```

Encode and decode with UTF-8

You don't need to worry about how Python stores each Unicode character when you do normal string processing.

However, when you exchange data with the outside world, you need a couple of things:

Scanned with CamScanner
Scanned with CamScanner

- A way to *encode* character strings to bytes
- A way to *decode* bytes to character strings

If there were fewer than 64,000 characters in Unicode, we could store each Unicode character ID in two bytes. Unfortunately, there are more. We could encode each ID in three or four bytes, but that would increase the memory and disk storage space needs for common text strings by three or four times.

Ken Thompson and Rob Pike, whose names will be familiar to Unix developers, designed the *UTF-8* dynamic encoding scheme one night on a placemat in a New Jersey diner. It uses one to four bytes per Unicode character:

- One byte for ASCII
- Two bytes for most Latin-derived (but not Cyrillic) languages
- Three bytes for the rest of the basic multilingual plane
- Four bytes for the rest, including some Asian languages and symbols

UTF-8 is the standard text encoding in Python, Linux, and HTML. It's fast, complete, and works well. If you use UTF-8 encoding throughout your code, life will be much easier than trying to hop in and out of various encodings.

> If you create a Python string by copying and pasting from another source such as a web page, be sure the source is encoded in the UTF-8 format. It's *very* common to see text that was encoded as Latin-1 or Windows 1252 copied into a Python string, which causes an exception later with an invalid byte sequence.

Encoding

You *encode* a *string* to *bytes*. The string `encode()` function's first argument is the encoding name. The choices include those presented in Table 7-1.

Table 7-1. Encodings

`'ascii'`	Good old seven-bit ASCII
`'utf-8'`	Eight-bit variable-length encoding, and what you almost always want to use
`'latin-1'`	Also known as ISO 8859-1
`'cp-1252'`	A common Windows encoding
`'unicode-escape'`	Python Unicode literal format, `\uxxxx` or `\Uxxxxxxxx`

You can encode anything as UTF-8. Let's assign the Unicode string `'\u2603'` to the name `snowman`:

```
>>> snowman = '\u2603'
```


Scanned with CamScanner
Scanned with CamScanner

snowman is a Python Unicode string with a single character, regardless of how many bytes might be needed to store it internally:

```
>>> len(snowman)
1
```

Next let's encode this Unicode character to a sequence of bytes:

```
>>> ds = snowman.encode('utf-8')
```

As I mentioned earlier, UTF-8 is a variable-length encoding. In this case, it used three bytes to encode the single snowman Unicode character:

```
>>> len(ds)
3
>>> ds
b'\xe2\x98\x83'
```

Now, `len()` returns the number of bytes (3) because `ds` is a `bytes` variable.

You can use encodings other than UTF-8, but you'll get errors if the Unicode string can't be handled by the encoding. For example, if you use the `ascii` encoding, it will fail unless your Unicode characters happen to be valid ASCII characters as well:

```
>>> ds = snowman.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\u2603'
in position 0: ordinal not in range(128)
```

The `encode()` function takes a second argument to help you avoid encoding exceptions. Its default value, which you can see in the previous example, is `'strict'`; it raises a `UnicodeEncodeError` if it sees a non-ASCII character. There are other encodings. Use `'ignore'` to throw away anything that won't encode:

```
>>> snowman.encode('ascii', 'ignore')
b''
```

Use `'replace'` to substitute `?` for unknown characters:

```
>>> snowman.encode('ascii', 'replace')
b'?'
```

Use `'backslashreplace'` to produce a Python Unicode character string, like `unicode-escape`:

```
>>> snowman.encode('ascii', 'backslashreplace')
b'\\u2603'
```

You would use this if you needed a printable version of the Unicode escape sequence.

The following produces character entity strings that you can use in web pages:

Scanned with CamScanner
Scanned with CamScanner

```
>>> snowman.encode('ascii', 'xmlcharrefreplace')
b'&#9731;'
```

Decoding

We *decode* byte strings to Unicode strings. Whenever we get text from some external source (files, databases, websites, network APIs, and so on), it's encoded as byte strings. The tricky part is knowing which encoding was actually used, so we can *run it backward* and get Unicode strings.

The problem is that nothing in the byte string itself says what encoding was used. I mentioned the perils of copying and pasting from websites earlier. You've probably visited websites with odd characters where plain old ASCII characters should be.

Let's create a Unicode string called `place` with the value `'café'`:

```
>>> place = 'caf\u00e9'
>>> place
'café'
>>> type(place)
<class 'str'>
```

Encode it in UTF-8 format in a `bytes` variable called `place_bytes`:

```
>>> place_bytes = place.encode('utf-8')
>>> place_bytes
b'caf\xc3\xa9'
>>> type(place_bytes)
<class 'bytes'>
```

Notice that `place_bytes` has five bytes. The first three are the same as ASCII (a strength of UTF-8), and the final two encode the `'é'`. Now, let's decode that byte string back to a Unicode string:

```
>>> place2 = place_bytes.decode('utf-8')
>>> place2
'café'
```

This worked because we encoded to UTF-8 and decoded from UTF-8. What if we told it to decode from some other encoding?

```
>>> place3 = place_bytes.decode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 3:
ordinal not in range(128)
```

The ASCII decoder threw an exception because the byte value `0xc3` is illegal in ASCII. There are some 8-bit character set encodings in which values between 128 (hex 80) and 255 (hex FF) are legal but not the same as UTF-8:

```
>>> place4 = place_bytes.decode('latin-1')
>>> place4
```


Scanned with CamScanner
Scanned with CamScanner

```
'cafÃ©'
>>> place5 = place_bytes.decode('windows-1252')
>>> place5
'cafÃ©'
```

Urk.

The moral of this story: whenever possible, use UTF-8 encoding. It works, is supported everywhere, can express every Unicode character, and is quickly decoded and encoded.

For more information

If you would like to learn more, these links are particularly helpful:

- Unicode HOWTO
- Pragmatic Unicode
- The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)

Format

We've pretty much ignored text formatting—until now. Chapter 2 shows a few string alignment functions, and the code examples have used simple `print()` statements, or just let the interactive interpreter display values. But it's time we look at how to *interpolate* data values into strings—in other words, put the values inside the strings using various formats. You can use this to produce reports and other outputs for which appearances need to be just so.

Python has two ways of formatting strings, loosely called *old style* and *new style*. Both styles are supported in Python 2 and 3 (new style in Python 2.6 and up). Old style is simpler, so we'll begin there.

Old style with %

The old style of string formatting has the form *`string % data`*. Inside the string are interpolation sequences. Table 7-2 illustrates that the very simplest sequence is a % followed by a letter indicating the data type to be formatted.

Scanned with CamScanner
Scanned with CamScanner

There are a few cases in which the regular expression pattern rules conflict with the Python string rules. The following pattern should match any word that begins with `fish`:

```
>>> re.findall('\bfish', source)
[]
```

Why doesn't it? As is discussed in Chapter 2, Python employs a few special *escape characters* for strings. For example, `\b` means backspace in strings, but in the mini-language of regular expressions it means the beginning of a word. Avoid the accidental use of escape characters by using Python's *raw strings* when you define your regular expression string. Always put an `r` character before your regular expression pattern string, and Python escape characters will be disabled, as demonstrated here:

```
>>> re.findall(r'\bfish', source)
['fish']
```

Patterns: specifying match output

When using `match()` or `search()`, all matches are returned from the result object `m` as `m.group()`. If you enclose a pattern in parentheses, the match will be saved to its own group, and a tuple of them will be available as `m.groups()`, as shown here:

```
>>> m = re.search(r'(. dish\b).*(\bfish)', source)
>>> m.group()
'a dish of fish'
>>> m.groups()
('a dish', 'fish')
```

If you use this pattern `(?P< name > expr )`, it will match *expr*, saving the match in group *name*:

```
>>> m = re.search(r'(?P<DISH>. dish\b).*(?P<FISH>\bfish)', source)
>>> m.group()
'a dish of fish'
>>> m.groups()
('a dish', 'fish')
>>> m.group('DISH')
'a dish'
>>> m.group('FISH')
'fish'
```

Binary Data

Text data can be challenging, but binary data can be, well, interesting. You need to know about concepts such as *endianness* (how your computer's processor breaks data into bytes) and *sign bits* for integers. You might need to delve into binary file formats or network packets to extract or even change data. This section will show you the basics of binary data wrangling in Python.

bytes and bytearray

Python 3 introduced the following sequences of eight-bit integers, with possible values from 0 to 255, in two types:

- *bytes* is immutable, like a tuple of bytes
- *bytearray* is mutable, like a list of bytes

Beginning with a list called `blist`, this next example creates a bytes variable called `the_bytes` and a `bytearray` variable called `the_byte_array`:

```
>>> blist = [1, 2, 3, 255]
>>> the_bytes = bytes(blist)
>>> the_bytes
b'\x01\x02\x03\xff'
>>> the_byte_array = bytearray(blist)
>>> the_byte_array
bytearray(b'\x01\x02\x03\xff')
```

> The representation of a bytes value begins with a b and a quote character, followed by hex sequences such as \x02 or ASCII characters, and ends with a matching quote character. Python converts the hex sequences or ASCII characters to little integers, but shows byte values that are also valid ASCII encodings as ASCII characters.
>
> ```
> >>> b'\x61'
> b'a'
>
> >>> b'\x01abc\xff'
> b'\x01abc\xff'
> ```

This next example demonstrates that you can't change a bytes variable:

```
>>> the_bytes[1] = 127
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
```

But a bytearray variable is mellow and mutable:

```
>>> the_byte_array = bytearray(blist)
>>> the_byte_array
bytearray(b'\x01\x02\x03\xff')
>>> the_byte_array[1] = 127
>>> the_byte_array
bytearray(b'\x01\x7f\x03\xff')
```

Each of these would create a 256-element result, with values from 0 to 255:

```
>>> the_bytes = bytes(range(0, 256))
>>> the_byte_array = bytearray(range(0, 256))
```


Scanned with CamScanner
Scanned with CamScanner

When printing bytes or bytearray data, Python uses \x*xx* for non-printable bytes and their ASCII equivalents for printable ones (plus some common escape characters, such as \n instead of \x0a). Here's the printed representation of the_bytes (manually reformatted to show 16 bytes per line):

```
>>> the_bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f
\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
!"#$%&\'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~\x7f
\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f
\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f
\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf
\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf
\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf
\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf
\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef
\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff'
```

This can be confusing, because they're bytes (teeny integers), not characters.

Convert Binary Data with struct

As you've seen, Python has many tools for manipulating text. Tools for binary data are much less prevalent. The standard library contains the struct module, which handles data similar to *structs* in C and C++. Using struct, you can convert binary data to and from Python data structures.

Let's see how this works with data from a PNG file—a common image format that you'll see along with GIF and JPEG files. We'll write a small program that extracts the width and height of an image from some PNG data.

We'll use the O'Reilly logo—the little bug-eyed tarsier shown in Figure 7-1.

Figure 7-1. The O'Reilly tarsier

Scanned with CamScanner
Scanned with CamScanner
Scanned with CamScanner

The PNG file for this image is available on Wikipedia. We don't show how to read files until Chapter 8, so I downloaded this file, wrote a little program to print its values as bytes, and just typed the values of the first 30 bytes into a Python bytes variable called data for the example that follows. (The PNG format specification stipulates that the width and height are stored within the first 24 bytes, so we don't need more than that for now.)

```
>>> import struct
>>> valid_png_header = b'\x89PNG\r\n\x1a\n'
>>> data = b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR' + \
...     b'\x00\x00\x00\x9a\x00\x00\x00\x8d\x08\x02\x00\x00\x00\xc0'
>>> if data[:8] == valid_png_header:
...     width, height = struct.unpack('>LL', data[16:24])
...     print('Valid PNG, width', width, 'height', height)
... else:
...     print('Not a valid PNG')
...
Valid PNG, width 154 height 141
```

Here's what this code does:

- data contains the first 30 bytes from the PNG file. To fit on the page, I joined two byte strings with + and the continuation character (\).
- valid_png_header contains the 8-byte sequence that marks the start of a valid PNG file.
- width is extracted from bytes 16-20, and height from bytes 21-24.

The >LL is the format string that instructs unpack() how to interpret its input byte sequences and assemble them into Python data types. Here's the breakdown:

- The > means that integers are stored in *big-endian* format.
- Each L specifies a 4-byte unsigned long integer.

You can examine each 4-byte value directly:

```
>>> data[16:20]
b'\x00\x00\x00\x9a'
>>> data[20:24]
b'\x00\x00\x00\x8d'
```

Big-endian integers have the most significant bytes to the left. Because the width and height are each less than 255, they fit into the last byte of each sequence. You can verify that these hex values match the expected decimal values:

```
>>> 0x9a
154
>>> 0x8d
141
```


Scanned with CamScanner
Scanned with CamScanner

When you want to go in the other direction and convert Python data to bytes, use the struct pack() function:

```
>>> import struct
>>> struct.pack('>L', 154)
b'\x00\x00\x00\x9a'
>>> struct.pack('>L', 141)
b'\x00\x00\x00\x8d'
```

Table 7-5 and Table 7-6 show the format specifiers for pack() and unpack().

The endian specifiers go first in the format string.

Table 7-5. Endian specifiers

Specifier	Byte order
<	little endian
>	big endian

Table 7-6. Format specifiers

Specifier	Description	Bytes
x	skip a byte	1
b	signed byte	1
B	unsigned byte	1
h	signed short integer	2
H	unsigned short integer	2
i	signed integer	4
I	unsigned integer	4
l	signed long integer	4
L	unsigned long integer	4
Q	unsigned long long integer	8
f	single precision float	4
d	double precision float	8
p	*count* and characters	1 + *count*
s	characters	*count*

The type specifiers follow the endian character. Any specifier may be preceded by a number that indicates the *count*; 5B is the same as BBBBB.

You can use a *count* prefix instead of >LL:

```
>>> struct.unpack('>2L', data[16:24])
(154, 141)
```


Scanned with CamScanner
Scanned with CamScanner

We used the slice `data[16:24]` to grab the interesting bytes directly. We could also use the x specifier to skip the uninteresting parts:

```
>>> struct.unpack('>16x2L6x', data)
(154, 141)
```

This means:

- Use big-endian integer format (>)
- Skip 16 bytes (16x)
- Read 8 bytes—two unsigned long integers (2L)
- Skip the final 6 bytes (6x)

Other Binary Data Tools

Some third-party open source packages offer the following, more declarative ways of defining and extracting binary data:

- bitstring
- construct
- hachoir
- binio

Appendix D has details on how to download and install external packages such as these. For the next example, you need to install `construct`. Here's all you need to do:

```
$ pip install construct
```

Here's how to extract the PNG dimensions from our data bytestring by using `construct`:

```
>>> from construct import Struct, Magic, UBInt32, Const, String
>>> # adapted from code at https://github.com/construct
>>> fmt = Struct('png',
...     Magic(b'\x89PNG\r\n\x1a\n'),
...     UBInt32('length'),
...     Const(String('type', 4), b'IHDR'),
...     UBInt32('width'),
...     UBInt32('height')
...     )
>>> data = b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR' + \
...     b'\x00\x00\x00\x9a\x00\x00\x00\x8d\x08\x02\x00\x00\x00\xc0'
>>> result = fmt.parse(data)
>>> print(result)
Container:
    length = 13
```

```
    type = b'IHDR'
    width = 154
    height = 141
>>> print(result.width, result.height)
154, 141
```

Convert Bytes/Strings with binascii()

The standard `binascii` module has functions with which you can convert between binary data and various string representations: hex (base 16), base 64, uuencoded, and others. For example, in the next snippet, let's print that 8-byte PNG header as a sequence of hex values, instead of the mixture of ASCII and `\x xx` escapes that Python uses to display *bytes* variables:

```
>>> import binascii
>>> valid_png_header = b'\x89PNG\r\n\x1a\n'
>>> print(binascii.hexlify(valid_png_header))
b'89504e470d0a1a0a'
```

Hey, this thing works backwards, too:

```
>>> print(binascii.unhexlify(b'89504e470d0a1a0a'))
b'\x89PNG\r\n\x1a\n'
```

Bit Operators

Python provides bit-level integer operators, similar to those in the C language. Table 7-7 summarizes them and includes examples with the integers `a` (decimal 5, binary `0b0101`) and `b` (decimal 1, binary `0b0001`).

Table 7-7. Bit-level integer operators

Operator	Description	Example	Decimal result	Binary result
&	and	a & b	1	0b0001
|	or	a | b	5	0b0101
^	exclusive or	a ^ b	4	0b0100
~	flip bits	~a	-6	*binary representation depends on int size*
<<	left shift	a << 1	10	0b1010
>>	right shift	a >> 1	2	0b0010

These operators work something like the set operators in Chapter 3. The & operator returns bits that are the same in both arguments, and | returns bits that are set in either of them. The ^ operator returns bits that are in one or the other, but not both. The ~ operator reverses all the bits in its single argument; this also reverses the sign because an integer's highest bit indicates its sign (1 = negative) in *two's complement* arithmetic, used in all modern computers. The << and >> operators just move bits to the left or right.

Scanned with CamScanner
Scanned with CamScanner

A left shift of one bit is the same as multiplying by two, and a right shift is the same as dividing by two.

Things to Do

7.1. Create a Unicode string called `mystery` and assign it the value `'\U0001f4a9'`. Print `mystery`. Look up the Unicode name for `mystery`.

7.2. Encode `mystery`, this time using UTF-8, into the `bytes` variable `pop_bytes`. Print `pop_bytes`.

7.3. Using UTF-8, decode `pop_bytes` into the string variable `pop_string`. Print `pop_string`. Is `pop_string` equal to `mystery`?

7.4. Write the following poem by using old-style formatting. Substitute the strings `'roast beef'`, `'ham'`, `'head'`, and `'clam'` into this string:

```
My kitty cat likes %s,
My kitty cat likes %s,
My kitty cat fell on his %s
And now thinks he's a %s.
```

7.5. Write a form letter by using new-style formatting. Save the following string as `letter` (you'll use it in the next exercise):

```
Dear {salutation} {name},

Thank you for your letter. We are sorry that our {product} {verbed} in your
{room}. Please note that it should never be used in a {room}, especially
near any {animals}.

Send us your receipt and {amount} for shipping and handling. We will send
you another {product} that, in our tests, is {percent}% less likely to
have {verbed}.

Thank you for your support.

Sincerely,
{spokesman}
{job_title}
```

7.6. Make a dictionary called `response` with values for the string keys `'salutation'`, `'name'`, `'product'`, `'verbed'` (past tense verb), `'room'`, `'animals'`, `'amount'`, `'percent'`, `'spokesman'`, and `'job_title'`. Print `letter` with the values from `response`.

7.7. When you're working with text, regular expressions come in very handy. We'll apply them in a number of ways to our featured text sample. It's a poem titled "Ode on the Mammoth Cheese," written by James McIntyre in 1866 in homage to a seven-thousand-pound cheese that was crafted in Ontario and sent on an international tour. If you'd rather not type all of it, use your favorite search engine and cut and paste the words into

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 241.

> 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')]
```


Scanned with CamScanner
Scanned with CamScanner

```
...     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()
```


Scanned with CamScanner
Scanned with CamScanner

```
>>> 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 `readlines()` 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
```


Scanned with CamScanner
Scanned with CamScanner

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)
...
```


Scanned with CamScanner
Scanned with CamScanner

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()
[illegible]
```

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

```
>>> fin.seek([illegible])
[illegible]
```

Read until the end of the file:

```
>>> bdata = fin.read()
>>> len(bdata)

>>> bdata[[illegible]]
[illegible]
```

`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 [illegible]
>>> os.SEEK_SET
[illegible]
>>> os.SEEK_CUR
[illegible]
>>> os.SEEK_END
[illegible]
```

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

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


Scanned with CamScanner
Scanned with CamScanner

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.

Scanned with CamScanner
Scanned with CamScanner

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 81 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:

Scanned with CamScanner
Scanned with CamScanner

```
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

20(69 Com

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>
```


Scanned with CamScanner

Scanned with CamScanner

```
  </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)
```


Scanned with CamScanner
Scanned with CamScanner

```
...     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.

Scanned with CamScanner
Scanned with CamScanner

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.

Scanned with CamScanner
Scanned with CamScanner

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 250), 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 126. 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 `datetime.datetime`. Because everything in Python is an object, `isinstance()` works everywhere:

Scanned with CamScanner
Scanned with CamScanner

```
>>> 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,
```


Scanned with CamScanner
Scanned with CamScanner

```
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;">
```


Scanned with CamScanner
Scanned with CamScanner

```
  <!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 [illegible]>
>>> cfg['french']
```


Scanned with CamScanner
Scanned with CamScanner

```
<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 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 a couple of modules that you might find useful:

- `h5py` is a full-featured low-level interface. Read the documentation and code.

Scanned with CamScanner
Scanned with CamScanner

- 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 Appendix C. 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 format.

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 *tables* (as they are usually called nowadays). For instance, in our menu example earlier, there is a relationship between each item and its price.

A table is a grid of rows and columns, similar to a spreadsheet. 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 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.

Scanned with CamScanner
Scanned with CamScanner

> 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 non-key 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, that a client sends to the database server, which figures out what to do with them.

There have been various SQL standard definitions, but 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 uses

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

Table 8-1 lists the basic SQL DDL commands.

Table 8-1. 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`

Scanned with CamScanner
Scanned with CamScanner

> Why all the CAPITAL LETTERS? SQL is case-insensitive, but it's 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:

- Create by using the SQL `INSERT` statement
- Read by using `SELECT`
- Update by using `UPDATE`
- Delete by using `DELETE`

Table 8-2 looks at the commands available for SQL DML.

Table 8-2. 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.

Scanned with CamScanner
Scanned with CamScanner

`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, smart phones, 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.

Scanned with CamScanner
Scanned with CamScanner

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 list to the `execute()` function. Placeholders handle tedious details such as quoting. They protect you against *SQL injection*—a kind of external attack that is common on the Web that inserts malicious SQL commands into the system.

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:

Scanned with CamScanner
Scanned with CamScanner

```
>>> 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.

MysqlDB has been the most popular MySQL driver, but it has not yet been ported to Python 3. Table 8-3 lists the drivers you can use to access MySQL from Python.

Table 8-3. MySQL drivers

Name	Link	Pypi package	Import as	Notes
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, in many ways more advanced than MySQL. Table 8-4 presents the Python drivers you can use to access it.

Table 8-4. 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	*http://python.projects.pgfoundry.org/*	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
```


Scanned with CamScanner
Scanned with CamScanner

You can use SQLAlchemy on several levels:

- The lowest level handles database connection *pools*, executing SQL commands, and returning results. This is closest to the DB-API.
- Next up is the *SQL expression language*, a Pythonic SQL builder.
- 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 only needed if it's not the standard one for this server)

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

Table 8-5 lists the dialects and drivers.

Table 8-5. SQLAlchemy connection

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

Scanned with CamScanner
Scanned with CamScanner

The engine layer

First, we'll 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 into Python. The connection string for SQLite skips the *host*, *port*, *user*, and *password*. The *dbname* informs SQLite as to 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 OS X; 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')
```


Scanned with CamScanner
Scanned with CamScanner

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 [illegible]>
```

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'll 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.

Scanned with CamScanner
Scanned with CamScanner

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

In the last 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'll 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')
```


Scanned with CamScanner
Scanned with CamScanner

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, [illegible])
>>> third = Zoo('weasel', 1, [illegible])
>>> 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 [illegible] 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 [illegible]
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .tables
zoo
sqlite> select * from [illegible];
```


Scanned with CamScanner
Scanned with CamScanner

```
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 room
- 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 break down at some point; they're leaky. 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 sparingly, and mostly for simple applications. If the application is that simple, maybe you can just use straight SQL (or the SQL Expression Language), anyhow.

Or, you can try something simpler such as `dataset`. It's built on SQLAlchemy and provides a simple ORM for SQL, JSON, and CSV storage.

NoSQL Data Stores

Some databases are not relational and don't support SQL. These were written to handle very large data sets, allow more flexible data definitions, or support custom data operations. They've been collectively labeled *NoSQL* (formerly meaning *no SQL*; now the less confrontational *not only SQL*).

The dbm Family

The `dbm` formats were around long before *NoSQL* was coined. They're *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 get a value from a key.

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:

Scanned with CamScanner
Scanned with CamScanner

```
>>> 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 if 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()`. Note that `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 OS X, and for Windows. If you want to try out this section, you'll need a 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 is *not* persistent, and data that you wrote earlier might disappear. This is inherent in memcached, being that it's a cache server. 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:

Scanned with CamScanner
Scanned with CamScanner

```
>>> 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*. Like memcached, all of the data in a Redis server should fit in memory (although there is now an option to save the data to disk). 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 online documentation. You can install it by using this command:

```
$ pip install redis
```

The Redis server itself 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()
```

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

Scanned with CamScanner
Scanned with CamScanner

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 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=start, -1=end):

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

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

```
>>> conn.setrange('secret', 0, 'ICKY')

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

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

Scanned with CamScanner
Scanned with CamScanner

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
```


Scanned with CamScanner
Scanned with CamScanner

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')
```

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 10 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. Thus, 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')
```

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()`:

Scanned with CamScanner
Scanned with CamScanner

```
>>> 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 can see in the series of examples that follow.

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 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:

Scanned with CamScanner
Scanned with CamScanner

```
>>> 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'll show the last use case, tracking user logins via timestamps. We're using the Unix *epoch* value (more on this in Chapter 10) that's returned by the Python time() function:

```
>>> import
>>> now = time.time()
>>> now
[illegible]
```


Scanned with CamScanner
Scanned with CamScanner

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)]
```

Bits

This is a very space-efficient and fast way to deal with large sets of numbers. Suppose that you have a website with registered users. You'd like to track how often people log in, how many users visit on a particular day, how often the same user visits on following days, and so on. You could use Redis sets, but if you've assigned increasing numeric user IDs, bits are more compact and faster.

Let's begin by creating a bitset for each day. For this test, we'll just use three days and a few user IDs:

```
>>> days = ['2013-02-25', '2013-02-26', '2013-02-27']
>>> big_spender = 1089
>>> tire_kicker = 40459
>>> late_joiner = 550212
```


Scanned with CamScanner
Scanned with CamScanner

Each date is a separate key. Set the bit for a particular user ID for that date. For example, on the first date (2013-02-25), we had visits from big_spender (ID 1089) and tire_kicker (ID 40459):

```
>>> conn.setbit(days[0], big_spender, 1)
0
>>> conn.setbit(days[0], tire_kicker, 1)
0
```

The next day, big_spender came back:

```
>>> conn.setbit(days[1], big_spender, 1)
0
```

The next day had yet another visit from our friend, big_spender, and a new person whom we're calling late_joiner:

```
>>> conn.setbit(days[2], big_spender, 1)
0
>>> conn.setbit(days[2], late_joiner, 1)
0
```

Let's get the daily visitor count for these three days:

```
>>> for day in days:
...     conn.bitcount(day)
...
2
1
2
```

Did a particular user visit on a particular day?

```
>>> conn.getbit(days[1], tire_kicker)
0
```

So, tire_kicker did not visit on the second day.

How many users visited every day?

```
>>> conn.bitop('and', 'everyday', *days)
68777
>>> conn.bitcount('everyday')
1
```

I'll give you three guesses who it was:

```
>>> conn.getbit('everyday', big_spender)
1
```

Finally, what was the number of total unique users in these three days?

```
>>> conn.bitop('or', 'alldays', *days)
68777
>>> conn.bitcount('alldays')
3
```


Scanned with CamScanner
Scanned with CamScanner

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. As is demonstrated here, 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.

Other NoSQL

The NoSQL servers listed here handle data larger than memory, and many of them use multiple computers. Table 8-6 presents notable servers and their Python libraries.

Table 8-6. NoSQL databases

Site	Python API
Cassandra	pycassa
CouchDB	couchdb-python
HBase	happybase
Kyoto Cabinet	kyotocabinet
MongoDB	mongodb
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, and their Python APIs, in Table 8-7.

Table 8-7. Full-text databases

Site	Python API
Lucene	pylucene

Scanned with CamScanner
Scanned with CamScanner