In Python, some names are spelled in a peculiar manner, with two leading and two trailing underscores. You have already encountered some of these (\_future\_, for example). This spelling signals that the name has a special significance—you should never invent such names for your own programs. One very prominent set of such names in the language consists of the magic (or special) method names. If your object implements one of these methods, that method will be called under specific circumstances (exactly which will depend on the name) by Python. There is rarely any need to call these methods directly.

This chapter deals with a few important magic methods (most notably the \_init\_ method and some methods dealing with item access, allowing you to create sequences or mappings of your own). It also tackles two related topics: properties (dealt with through magic methods in previous versions of Python, but now handled by the property function), and iterators (which use the magic method \_iter\_ to enable them to be used in for loops). You’ll find a meaty example at the end of the chapter, which uses some of the things you have learned so far to solve a fairly difficult problem.

**Before We Begin . . .**

A while ago (in version 2.2), the way Python objects work changed quite a bit. This change has several consequences, most of which won’t be important to you as a beginning Python programmer.\(^1\) One thing is worth noting, though: even if you’re using a recent version of Python, some features (such as properties and the super function) won’t work on “old-style” classes. To make your classes “new-style,” you should either put the assignment \_metaclass\_ = type at the top of your modules (as mentioned in Chapter 7) or (directly or indirectly) subclass the built-in class (or, actually, type) object (or some other new-style class). Consider the following two classes:

```python
class NewStyle(object):
    more_code_here
```

---

\(^1\) For a thorough description of the differences between old-style and new-style classes, see Chapter 8 in Alex Martelli’s *Python in a Nutshell* (O’Reilly & Associates, 2003).
class OldStyle:
    more_code_here

    Of these two, NewStyle is a new-style class; OldStyle is an old-style class. If the file began with `__metaclass__ = type`, though, both classes would be new-style.

---

**Note** You can also assign to the `__metaclass__` variable in the class scope of your class. That would set the metaclass of only that class. Metaclasses are the classes of other classes (or types)—a rather advanced topic. For more information about metaclasses, take a look at the (somewhat technical) article called “Unifying types and classes in Python 2.2” by Guido van Rossum (http://python.org/2.2/descript.html), or do a web search for the term “python metaclasses.”

---

I do not explicitly set the metaclass (or subclass object) in all the examples in this book. However, if you do not specifically need to make your programs compatible with old versions of Python, I advise you to make all your classes new-style, and consistently use features such as the `super` function (described in the section “Using the `super` Function,” later in this chapter).

---

**Note** There are no “old-style” classes in Python 3.0, and no need to explicitly subclass object or set the metaclass to `type`. All classes will implicitly be subclasses of object—directly, if you don’t specify a superclass, or indirectly otherwise.

---

**Constructors**

The first magic method we’ll take a look at is the constructor. In case you have never heard the word *constructor* before, it’s basically a fancy name for the kind of initializing method I have already used in some of the examples, under the name `init`. What separates constructors from ordinary methods, however, is that the constructors are called automatically right after an object has been created. Thus, instead of doing what I’ve been doing up until now:

```python
>>> f = FooBar()
>>> f.init()
```

constructors make it possible to simply do this:

```python
>>> f = FooBar()
```