CHAPTER 9

# Magic Methods, Properties, and Iterators

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:

```
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/descrintro.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:

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

constructors make it possible to simply do this:

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

Creating constructors in Python is really easy; simply change the init method's name from the plain old init to the magic version, __init__:

```
class FooBar:
    def __init__(self):
        self.somevar = 42

>>> f = FooBar()
>>> f.somevar
42
```

Now, that's pretty nice. But you may wonder what happens if you give the constructor some parameters to work with. Consider the following:

```
class FooBar:
    def __init__(self, value=42):
        self.somevar = value
```

How do you think you could use this? Because the parameter is optional, you certainly could go on like nothing had happened. But what if you wanted to use it (or you hadn't made it optional)? I'm sure you've guessed it, but let me show you anyway:

```
>>> f = FooBar('This is a constructor argument')
>>> f.somevar
'This is a constructor argument'
```

Of all the magic methods in Python, __init__ is quite certainly the one you'll be using the most.

■ **Note** Python has a magic method called __del__, also known as the *destructor*. It is called just before the object is destroyed (garbage-collected), but because you cannot really know when (or if) this happens, I advise you to stay away from __del__ if at all possible.

## Overriding Methods in General, and the Constructor in Particular

In Chapter 7, you learned about inheritance. Each class may have one or more superclasses, from which they inherit behavior. If a method is called (or an attribute is accessed) on an instance of class B and it is not found, its superclass A will be searched. Consider the following two classes:

```
class A:
    def hello(self):
        print "Hello, I'm A."
```

```
class B(A):
    pass
```

Class A defines a method called hello, which is inherited by class B. Here is an example of how these classes work:

```
>>> a = A()
>>> b = B()
>>> a.hello()
Hello, I'm A.
>>> b.hello()
Hello, I'm A.
```

Because B does not define a hello method of its own, the original message is printed when hello is called.

One basic way of adding functionality in the subclass is simply to add methods. However, you may want to customize the inherited behavior by overriding some of the superclass's methods. For example, it is possible for B to override the hello method. Consider this modified definition of B:

```
class B(A):
    def hello(self):
        print "Hello, I'm B."
```

Using this definition, b.hello() will give a different result:

```
>>> b = B()
>>> b.hello()
Hello, I'm B.
```

Overriding is an important aspect of the inheritance mechanism in general, and may be especially important for constructors. Constructors are there to initialize the state of the newly constructed object, and most subclasses will need to have initialization code of their own, in addition to that of the superclass. Even though the mechanism for overriding is the same for all methods, you will most likely encounter one particular problem more often when dealing with constructors than when overriding ordinary methods: if you override the constructor of a class, you need to call the constructor of the superclass (the class you inherit from) or risk having an object that isn't properly initialized.

Consider the following class, Bird:

```
class Bird:
    def __init__(self):
        self.hungry = True
    def eat(self):
        if self.hungry:
            print 'Aaaah...'
            self.hungry = False
        else:
            print 'No, thanks!'
```

This class defines one of the most basic capabilities of all birds: eating. Here is an example of how you might use it:

```
>>> b = Bird()
>>> b.eat()
Aaaah...
>>> b.eat()
No, thanks!
```

As you can see from this example, once the bird has eaten, it is no longer hungry. Now consider the subclass SongBird, which adds singing to the repertoire of behaviors:

```
class SongBird(Bird):
    def __init__(self):
        self.sound = 'Squawk!'
    def sing(self):
        print self.sound
```

The SongBird class is just as easy to use as Bird:

```
>>> sb = SongBird()
>>> sb.sing()
Squawk!
```

Because SongBird is a subclass of Bird, it inherits the eat method, but if you try to call it, you'll discover a problem:

```
>>> sb.eat()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "birds.py", line 6, in eat
    if self.hungry:
AttributeError: SongBird instance has no attribute 'hungry'
```

The exception is quite clear about what's wrong: the SongBird has no attribute called hungry. Why should it? In SongBird, the constructor is overridden, and the new constructor doesn't contain any initialization code dealing with the hungry attribute. To rectify the situation, the SongBird constructor must call the constructor of its superclass, Bird, to make sure that the basic initialization takes place. There are basically two ways of doing this: by calling the unbound version of the superclass's constructor or by using the super function. In the next two sections, I explain both techniques.

## Calling the Unbound Superclass Constructor

The approach described in this section is, perhaps, mainly of historical interest. With current versions of Python, using the super function (as explained in the following section) is clearly the way to go (and with Python 3.0, it will be even more so). However, much existing code uses the approach described in this section, so you need to know about it. Also, it can be quite instructive—it's a nice example of the difference between bound and unbound methods.

Now, let's get down to business. If you find the title of this section a bit intimidating, relax. Calling the constructor of a superclass is, in fact, very easy (and useful). I'll start by giving you the solution to the problem posed at the end of the previous section:

```
class SongBird(Bird):
    def __init__(self):
        Bird.__init__(self)
        self.sound = 'Squawk!'
    def sing(self):
        print self.sound
```

Only one line has been added to the `SongBird` class, containing the code `Bird.__init__(self)`. Before I explain what this really means, let me just show you that this really works:

```
>>> sb = SongBird()
>>> sb.sing()
Squawk!
>>> sb.eat()
Aaaah...
>>> sb.eat()
No, thanks!
```

But why does this work? When you retrieve a method from an instance, the `self` argument of the method is automatically *bound* to the instance (a so-called bound method). You've seen several examples of that. However, if you retrieve the method directly from the class (such as in `Bird.__init__`), there is no instance to which to bind. Therefore, you are free to supply any `self` you want to. Such a method is called *unbound*, which explains the title of this section.

By supplying the current instance as the `self` argument to the unbound method, the songbird gets the full treatment from its superclass's constructor (which means that it has its `hungry` attribute set).

## Using the super Function

If you're not stuck with an old version of Python, the `super` function is really the way to go. It works only with new-style classes, but you should be using those anyway. It is called with the current class and instance as its arguments, and any method you call on the returned object will be fetched from the superclass rather than the current class. So, instead of using `Bird` in the `SongBird` constructor, you can use `super(SongBird, self)`. Also, the `__init__` method can be called in a normal (bound) fashion.

■ **Note** In Python 3.0, `super` can be called without any arguments, and will do its job as if "by magic."

The following is an updated version of the bird example:

```
__metaclass__ = type # super only works with new-style classes

class Bird:
    def __init__(self):
        self.hungry = True
    def eat(self):
        if self.hungry:
            print 'Aaaah...'
            self.hungry = False
        else:
            print 'No, thanks!'

class SongBird(Bird):
    def __init__(self):
        super(SongBird, self).__init__()
        self.sound = 'Squawk!'
    def sing(self):
        print self.sound
```

This new-style version works just like the old-style one:

```
>>> sb = SongBird()
>>> sb.sing()
Squawk!
>>> sb.eat()
Aaaah...
>>> sb.eat()
No, thanks!
```

## WHAT'S SO SUPER ABOUT SUPER?

In my opinion, the `super` function is more intuitive than calling unbound methods on the superclass directly, but that is not its only strength. The `super` function is actually quite smart, so even if you have multiple superclasses, you only need to use `super` once (provided that all the superclass constructors also use `super`). Also, some obscure situations that are tricky when using old-style classes (for example, when two of your superclasses share a superclass) are automatically dealt with by new-style classes and `super`. You don't need to understand exactly how it works internally, but you should be aware that, in most cases, it is clearly superior to calling the unbound constructors (or other methods) of your superclasses.

So, what does `super` return, really? Normally, you don't need to worry about it, and you can just pretend it returns the superclass you need. What it actually does is return a *super object*, which will take care of method resolution for you. When you access an attribute on it, it will look through all your superclasses (and super-superclasses, and so forth until it finds the attribute (or raises an `AttributeError`).

## Item Access

Although `__init__` is by far the most important special method you'll encounter, many others are available to enable you to achieve quite a lot of cool things. One useful set of magic methods described in this section allows you to create objects that behave like sequences or mappings.

The basic sequence and mapping protocol is pretty simple. However, to implement all the functionality of sequences and mappings, there are many magic methods to implement. Luckily, there are some shortcuts, but I'll get to that.

■ **Note** The word *protocol* is often used in Python to describe the rules governing some form of behavior. This is somewhat similar to the notion of *interfaces* mentioned in Chapter 7. The protocol says something about which methods you should implement and what those methods should do. Because polymorphism in Python is based on only the object's behavior (and not on its *ancestry*, for example, its class or superclass, and so forth), this is an important concept: where other languages might require an object to belong to a certain class or to implement a certain interface, Python often simply requires it to follow some given protocol. So, to *be* a sequence, all you have to do is follow the sequence protocol.

### The Basic Sequence and Mapping Protocol

Sequences and mappings are basically collections of *items*. To implement their basic behavior (protocol), you need two magic methods if your objects are immutable, or four if they are mutable:

`__len__(self)`: This method should return the number of items contained in the collection. For a sequence, this would simply be the number of elements. For a mapping, it would be the number of key-value pairs. If `__len__` returns zero (and you don't implement `__nonzero__`, which overrides this behavior), the object is treated as *false* in a Boolean context (as with empty lists, tuples, strings, and dictionaries).

`__getitem__(self, key)`: This should return the value corresponding to the given key. For a sequence, the key should be an integer from zero to *n*–1 (or, it could be negative, as noted later), where *n* is the length of the sequence. For a mapping, you could really have any kind of keys.

`__setitem__(self, key, value)`: This should store `value` in a manner associated with `key`, so it can later be retrieved with `__getitem__`. Of course, you define this method only for mutable objects.

`__delitem__(self, key)`: This is called when someone uses the `del` statement on a part of the object, and should delete the element associated with `key`. Again, only mutable objects (and not all of them—only those for which you want to let items be removed) should define this method.

Some extra requirements are imposed on these methods:

- For a sequence, if the key is a negative integer, it should be used to count from the end. In other words, treat `x[-n]` the same as `x[len(x)-n]`.
- If the key is of an inappropriate type (such as a string key used on a sequence), a `TypeError` may be raised.
- If the index of a sequence is of the right type, but outside the allowed range, an `IndexError` should be raised.

Let's have a go at it—let's see if we can create an infinite sequence:

```
def checkIndex(key):
    """
    Is the given key an acceptable index?

    To be acceptable, the key should be a non-negative integer. If it
    is not an integer, a TypeError is raised; if it is negative, an
    IndexError is raised (since the sequence is of infinite length).
    """
    if not isinstance(key, (int, long)): raise TypeError
    if key<0: raise IndexError

class ArithmeticSequence:
    def __init__(self, start=0, step=1):
        """
        Initialize the arithmetic sequence.

        start   - the first value in the sequence
        step    - the difference between two adjacent values
        changed - a dictionary of values that have been modified by
                  the user
        """
        self.start = start                      # Store the start value
        self.step = step                        # Store the step value
        self.changed = {}                       # No items have been modified

    def __getitem__(self, key):
        """
        Get an item from the arithmetic sequence.
        """
        checkIndex(key)

        try: return self.changed[key]           # Modified?
        except KeyError:                        # otherwise...
            return self.start + key*self.step   # ...calculate the value
    def __setitem__(self, key, value):
        """
        Change an item in the arithmetic sequence.
        """
        checkIndex(key)

        self.changed[key] = value                # Store the changed value
```

This implements an *arithmetic sequence*—a sequence of numbers in which each is greater than the previous one by a constant amount. The first value is given by the constructor parameter `start` (defaulting to zero), while the step between the values is given by `step` (defaulting to one). You allow the user to change some of the elements by keeping the exceptions to the general rule in a dictionary called `changed`. If the element hasn't been changed, it is calculated as `self.start + key*self.step`.

Here is an example of how you can use this class:

```
>>> s = ArithmeticSequence(1, 2)
>>> s[4]
9
>>> s[4] = 2
>>> s[4]
2
>>> s[5]
11
```

Note that I want it to be illegal to delete items, which is why I haven't implemented `__del__`:

```
>>> del s[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: ArithmeticSequence instance has no attribute '__delitem__'
```

Also, the class has no `__len__` method because it is of infinite length.

If an illegal type of index is used, a `TypeError` is raised, and if the index is the correct type but out of range (that is, negative in this case), an `IndexError` is raised:

```
>>> s["four"]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "arithseq.py", line 31, in __getitem__
    checkIndex(key)
  File "arithseq.py", line 10, in checkIndex
    if not isinstance(key, int): raise TypeError
```

```
TypeError
>>> s[-42]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "arithseq.py", line 31, in __getitem__
    checkIndex(key)
  File "arithseq.py", line 11, in checkIndex
    if key<0: raise IndexError
IndexError
```

The index checking is taken care of by a utility function I've written for the purpose, `checkIndex`.

One thing that might surprise you about the `checkIndex` function is the use of `isinstance` (which you should rarely use because type or class checking goes against the grain of Python's polymorphism). I've used it because the language reference explicitly states that the index should be an integer (this includes long integers). And complying with standards is one of the (very few) valid reasons for using type checking.

---

■ **Note** You can simulate slicing, too, if you like. When slicing an instance that supports `__getitem__`, a slice object is supplied as the key. Slice objects are described in the Python Library Reference (`http://python.org/doc/lib`) in Section 2.1, "Built-in Functions," under the `slice` function. Python 2.5 also has the more specialized method called `__index__`, which allows you to use noninteger limits in your slices. This is mainly useful only if you wish to go beyond the basic sequence protocol, though.

---

## Subclassing list, dict, and str

While the four methods of the basic sequence/mapping protocol will get you far, the official language reference also recommends that several other magic and ordinary methods be implemented (see the section "Emulating container types" in the Python Reference Manual, `http://www.python.org/doc/ref/sequence-types.html`), including the `__iter__` method, which I describe in the section "Iterators," later in this chapter. Implementing all these methods (to make your objects fully polymorphically equivalent to lists or dictionaries) is a lot of work and hard to get right. If you want custom behavior in only *one* of the operations, it makes no sense that you should need to reimplement all of the others. It's just programmer laziness (also called common sense).

So what should you do? The magic word is *inheritance*. Why reimplement all of these things when you can inherit them? The standard library comes with three ready-to-use implementations of the sequence and mapping protocols (`UserList`, `UserString`, and `UserDict`), and in current versions of Python, you can subclass the built-in types themselves. (Note that this is mainly useful if your class's behavior is close to the default. If you need to reimplement most of the methods, it might be just as easy to write a new class.)

So, if you want to implement a sequence type that behaves similarly to the built-in lists, you can simply subclass list.

---

■ **Note** When you subclass a built-in type such as list, you are indirectly subclassing object. Therefore your class is automatically new-style, which means that features such as the super function are available.

---

Let's just do a quick example—a list with an access counter:

```
class CounterList(list):
    def __init__(self, *args):
        super(CounterList, self).__init__(*args)
        self.counter = 0
    def __getitem__(self, index):
        self.counter += 1
        return super(CounterList, self).__getitem__(index)
```

The CounterList class relies heavily on the behavior of its subclass superclass (list). Any methods not overridden by CounterList (such as append, extend, index, and so on) may be used directly. In the two methods that *are* overridden, super is used to call the superclass version of the method, adding only the necessary behavior of initializing the counter attribute (in __init__) and updating the counter attribute (in __getitem__).

---

■ **Note** Overriding __getitem__ is not a bulletproof way of trapping user access because there are other ways of accessing the list contents, such as through the pop method.

---

Here is an example of how CounterList may be used:

```
>>> cl = CounterList(range(10))
>>> cl
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> cl.reverse()
>>> cl
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> del cl[3:6]
>>> cl
[9, 8, 7, 3, 2, 1, 0]
>>> cl.counter
0
>>> cl[4] + cl[2]
9
>>> cl.counter
2
```

As you can see, `CounterList` works just like `list` in most respects. However, it has a `counter` attribute (initially zero), which is incremented each time you access a list element. After performing the addition `cl[4] + cl[2]`, the counter has been incremented twice, to the value 2.

## More Magic

Special (magic) names exist for many purposes—what I've shown you so far is just a small taste of what is possible. Most of the magic methods available are meant for fairly advanced use, so I won't go into detail here. However, if you are interested, it is possible to emulate numbers, make objects that can be called as if they were functions, influence how objects are compared, and much more. For more information about which magic methods are available, see section "Special method names" in the Python Reference Manual (`http://www.python.org/doc/ref/specialnames.html`).

## Properties

In Chapter 7, I mentioned *accessor methods*. Accessors are simply methods with names such as `getHeight` and `setHeight`, and are used to retrieve or rebind some attribute (which may be private to the class—see the section "Privacy Revisited" in Chapter 7). Encapsulating state variables (attributes) like this can be important if certain actions must be taken when accessing the given attribute. For example, consider the following `Rectangle` class:

```
class Rectangle:
    def __init__(self):
        self.width = 0
        self.height = 0
    def setSize(self, size):
        self.width, self.height = size
    def getSize(self):
        return self.width, self.height
```

Here is an example of how you can use the class:

```
>>> r = Rectangle()
>>> r.width = 10
>>> r.height = 5
>>> r.getSize()
(10, 5)
>>> r.setSize((150, 100))
>>> r.width
150
```

The `getSize` and `setSize` methods are accessors for a fictitious attribute called `size`—which is simply the tuple consisting of `width` and `height`. (Feel free to replace this with something more exciting, such as the area of the rectangle or the length of its diagonal.) This code isn't directly wrong, but it is flawed. The programmer using this class shouldn't need to worry about how it is implemented (encapsulation). If you someday wanted to change the implementation so that `size` was a real attribute and `width` and `height` were calculated on the fly, you would need to wrap *them* in accessors, and any programs using the class would also have to be rewritten. The client code (the code using your code) should be able to treat all your attributes in the same manner.

So what is the solution? Should you wrap all your attributes in accessors? That is a possibility, of course. However, it would be impractical (and kind of silly) if you had a lot of simple attributes, because you would need to write many accessors that did nothing but retrieve or set these attributes, with no useful action taken. This smells of *copy-paste* programming, or *cookie-cutter code,* which is clearly a bad thing (although quite common for this specific problem in certain languages). Luckily, Python can hide your accessors for you, making all of your attributes look alike. Those attributes that are defined through their accessors are often called *properties*.

Python actually has two mechanisms for creating properties in Python. I'll focus on the most recent one, the `property` function, which works only on new-style classes. Then I'll give you a short description of how to implement properties with magic methods.

## The property Function

Using the `property` function is delightfully simple. If you have already written a class such as `Rectangle` from the previous section, you need to add only a single line of code (in addition to subclassing `object`, or using `__metaclass__ = type`):

```
__metaclass__ = type

class Rectangle:
    def __init__(self):
        self.width = 0
        self.height = 0
    def setSize(self, size):
        self.width, self.height = size
    def getSize(self):
        return self.width, self.height
    size = property(getSize, setSize)
```

In this new version of `Rectangle`, a property is created with the `property` function with the accessor functions as arguments (the *getter* first, then the *setter*), and the name `size` is then bound to this property. After this, you no longer need to worry about how things are implemented, but can treat `width`, `height`, and `size` the same way:

```
>>> r = Rectangle()
>>> r.width = 10
>>> r.height = 5
>>> r.size
(10, 5)
>>> r.size = 150, 100
>>> r.width
150
```

As you can see, the `size` attribute is still subject to the calculations in `getSize` and `setSize`, but it looks just like a normal attribute.

---

■ **Note** If your properties are behaving oddly, make sure you're using a new-style class (by subclassing `object` either directly or indirectly—or by setting the metaclass directly). If you aren't, the *getter* part of the property will still work, but the *setter* may not (depending on your Python version). This can be a bit confusing.

---

In fact, the `property` function may be called with zero, one, three, or four arguments as well. If called without any arguments, the resulting property is neither readable nor writable. If called with only one argument (a getter method), the property is readable only. The third (optional) argument is a method used to *delete* the attribute (it takes no arguments). The fourth (optional) argument is a docstring. The parameters are called `fget`, `fset`, `fdel`, and `doc`—you can use them as keyword arguments if you want a property that, say, is only writable and has a docstring.

Although this section has been short (a testament to the simplicity of the `property` function), it is very important. The moral is this: with new-style classes, you should use `property` rather than accessors.

**BUT HOW DOES IT WORK?**

In case you're curious about how `property` does its magic, I'll give you an explanation here. If you don't care, just skip ahead.

The fact is that `property` isn't really a function—it's a class whose instances have some magic methods that do all the work. The methods in question are `__get__`, `__set__`, and `__delete__`. Together, these three methods define the so-called *descriptor protocol*. An object implementing any of these methods is a descriptor. The special thing about descriptors is how they are accessed. For example, when reading an attribute (specifically, when accessing it in an instance, but when the attribute is defined in the class), if the attribute is bound to an object that implements `__get__`, the object won't simply be returned; instead, the `__get__` method will be called and the resulting value will be returned. This is, in fact, the mechanism underlying properties, bound methods, static and class methods (see the following section for more information), and `super`. A brief description of the descriptor protocol may be found in the Python Reference Manual (`http://python.org/doc/ref/descriptors.html`). A more thorough source of information is Raymond Hettinger's How-To Guide for Descriptors (`http://users.rcn.com/python/download/Descriptor.htm`).

## Static Methods and Class Methods

Before discussing the old way of implementing properties, let's take a slight detour, and look at another couple of features that are implemented in a similar manner to the new-style properties. Static methods and class methods are created by wrapping methods in objects of the `staticmethod` and `classmethod` types, respectively. Static methods are defined without `self` arguments, and they can be called directly on the class itself. Class methods are defined with a self-like parameter normally called cls. You can call class methods directly on the class object too, but the cls parameter then automatically is bound to the class. Here is a simple example:

```
__metaclass__ = type

class MyClass:

    def smeth():
        print 'This is a static method'
    smeth = staticmethod(smeth)

    def cmeth(cls):
        print 'This is a class method of', cls
    cmeth = classmethod(cmeth)
```

The technique of wrapping and replacing the methods manually like this is a bit tedious. In Python 2.4, a new syntax was introduced for wrapping methods like this, called *decorators*. (They actually work with any callable objects as wrappers, and can be used on both methods and functions.) You specify one or more decorators (which are applied in reverse order) by listing them above the method (or function), using the @ operator:

```
__metaclass__ = type

class MyClass:

    @staticmethod
    def smeth():
        print 'This is a static method'

    @classmethod
    def cmeth(cls):
        print 'This is a class method of', cls
```

Once you've defined these methods, they can be used like this (that is, without instantiating the class):

```
>>> MyClass.smeth()
This is a static method
>>> MyClass.cmeth()
This is a class method of <class '__main__.MyClass'>
```

Static methods and class methods haven't historically been important in Python, mainly because you could always use functions or bound methods instead, in some way, but also because the support hasn't really been there in earlier versions. So even though you may not see them used much in current code, they do have their uses (such as factory functions, if you've heard of those), and you may well think of some new ones.

## __getattr__, __setattr__, and Friends

It's possible to intercept every attribute access on an object. Among other things, you could use this to implement properties with old-style classes (where `property` won't necessarily work as it should). To have code executed when an attribute is accessed, you must use a couple of magic methods. The following four provide all the functionality you need (in old-style classes, you only use the last three):

`__getattribute__(self, name)`: Automatically called when the attribute `name` is accessed. (This works correctly on new-style classes only.)

`__getattr__(self, name)`: Automatically called when the attribute `name` is accessed and the object has no such attribute.

`__setattr__(self, name, value)`: Automatically called when an attempt is made to bind the attribute `name` to `value`.

`__delattr__(self, name)`: Automatically called when an attempt is made to delete the attribute `name`.

Although a bit trickier to use (and in some ways less efficient) than `property`, these magic methods are quite powerful, because you can write code in one of these methods that deals with several properties. (If you have a choice, though, stick with `property`.)

Here is the `Rectangle` example again, this time with magic methods:

```
class Rectangle:
    def __init__(self):
        self.width = 0
        self.height = 0
    def __setattr__(self, name, value):
        if name == 'size':
            self.width, self.height = value
        else:
            self.__dict__[name] = value
    def __getattr__(self, name):
        if name == 'size':
            return self.width, self.height
        else:
            raise AttributeError
```

As you can see, this version of the class needs to take care of additional administrative details. When considering this code example, it's important to note the following:

- The `__setattr__` method is called even if the attribute in question is not `size`. Therefore, the method must take both cases into consideration: if the attribute is `size`, the same operation is performed as before; otherwise, the magic attribute `__dict__` is used. It contains a dictionary with all the instance attributes. It is used instead of ordinary attribute assignment to avoid having `__setattr__` called again (which would cause the program to loop endlessly).

- The `__getattr__` method is called only if a normal attribute is not found, which means that if the given name is not `size`, the attribute does not exist, and the method raises an `AttributeError`. This is important if you want the class to work correctly with built-in functions such as `hasattr` and `getattr`. If the name *is* `size`, the expression found in the previous implementation is used.

■**Note** Just as there is an "endless loop" trap associated with `__setattr__`, there is a trap associated with `__getattribute__`. Because it intercepts *all* attribute accesses (in new-style classes), it will intercept accesses to `__dict__` as well! The only safe way to access attributes on `self` inside `__getattribute__` is to use the `__getattribute__` method of the superclass (using `super`).

## Iterators

I've mentioned iterators (and iterables) briefly in preceding chapters. In this section, I go into some more detail. I cover only one magic method, `__iter__`, which is the basis of the iterator protocol.

### The Iterator Protocol

To *iterate* means to repeat something several times—what you do with loops. Until now I have iterated over only sequences and dictionaries in `for` loops, but the truth is that you can iterate over other objects, too: objects that implement the `__iter__` method.

The `__iter__` method returns an iterator, which is any object with a method called `next`, which is callable without any arguments. When you call the `next` method, the iterator should return its "next value." If the method is called, and the iterator has no more values to return, it should raise a `StopIteration` exception.

■**Note** The iterator protocol is changed a bit in Python 3.0. In the new protocol, iterator objects should have a method called `__next__` rather than `next`, and a new built-in function called `next` may be used to access this method. In other words, `next(it)` is the equivalent of the pre-3.0 `it.next()`.

What's the point? Why not just use a list? Because it may often be overkill. If you have a function that can compute values one by one, you may need them only one by one—not all at once, in a list, for example. If the number of values is large, the list may take up too much memory. But there are other reasons: using iterators is more general, simpler, and more elegant. Let's take a look at an example you couldn't do with a list, simply because the list would need to be of infinite length!

Our "list" is the sequence of Fibonacci numbers. An iterator for these could be the following:

```
class Fibs:
    def __init__(self):
        self.a = 0
        self.b = 1
    def next(self):
        self.a, self.b = self.b, self.a+self.b
        return self.a
    def __iter__(self):
        return self
```

Note that the iterator implements the `__iter__` method, which will, in fact, return the iterator itself. In many cases, you would put the `__iter__` method in *another* object, which you would use in the `for` loop. That would then return your iterator. It is recommended that iterators implement an `__iter__` method of their own in addition (returning `self`, just as I did here), so they themselves can be used directly in `for` loops.

**■Note** In formal terms, an object that implements the `__iter__` method is *iterable*, and the object implementing `next` is the *iterator*.

First, make a `Fibs` object:

```
>>> fibs = Fibs()
```

You can then use it in a `for` loop—for example, to find the smallest Fibonacci number that is greater than 1,000:

```
>>> for f in fibs:
        if f > 1000:
            print f
            break
...
1597
```

Here, the loop stops because I issue a `break` inside it; if I didn't, the `for` loop would never end.

**Tip** The built-in function `iter` can be used to get an iterator from an iterable object:

```
>>> it = iter([1, 2, 3])
>>> it.next()
1
>>> it.next()
2
```

It can also be used to create an iterable from a function or other callable object (see the Python Library Reference, `http://docs.python.org/lib/`, for details).

### Making Sequences from Iterators

In addition to *iterating* over the iterators and iterables (which is what you normally do), you can convert them to sequences. In most contexts in which you can use a sequence (except in operations such as indexing or slicing), you can use an iterator (or an iterable object) instead. One useful example of this is explicitly converting an iterator to a list using the `list` constructor:

```
>>> class TestIterator:
        value = 0
        def next(self):
            self.value += 1
            if self.value > 10: raise StopIteration
            return self.value
        def __iter__(self):
            return self
...
>>> ti = TestIterator()
>>> list(ti)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

## Generators

Generators (also called *simple generators* for historical reasons) are relatively new to Python, and are (along with iterators) perhaps one of the most powerful features to come along for years. However, the generator concept is rather advanced, and it may take a while before it “clicks” and you see how it works or how it would be useful for you. Rest assured that while generators can help you write really elegant code, you can certainly write any program you wish without a trace of generators.

A generator is a kind of iterator that is defined with normal function syntax. Exactly how generators work is best shown through example. Let's first have a look at how you make them and use them, and then take a peek under the hood.

## Making a Generator

Making a generator is simple; it's just like making a function. I'm sure you are starting to tire of the good old Fibonacci sequence by now, so let me do something else. I'll make a function that flattens nested lists. The argument is a list that may look something like this:

```
nested = [[1, 2], [3, 4], [5]]
```

In other words, it's a list of lists. My function should then give me the numbers in order. Here's a solution:

```
def flatten(nested):
    for sublist in nested:
        for element in sublist:
            yield element
```

Most of this function is pretty simple. First, it iterates over all the sublists of the supplied nested list; then it iterates over the elements of each sublist in order. If the last line had been `print element`, for example, the function would have been easy to understand, right?

So what's new here is the `yield` statement. Any function that contains a `yield` statement is called a *generator*. And it's not just a matter of naming; it will behave quite differently from ordinary functions. The difference is that instead of returning *one* value, as you do with `return`, you can yield *several* values, one at a time. Each time a value is yielded (with `yield`), the function *freezes*; that is, it stops its execution at exactly that point and waits to be reawakened. When it is, it resumes its execution at the point where it stopped.

I can make use of all the values by iterating over the generator:

```
>>> nested = [[1, 2], [3, 4], [5]]
>>> for num in flatten(nested):
        print num
...
1
2
3
4
5
```

or

```
>>> list(flatten(nested))
[1, 2, 3, 4, 5]
```

**LOOPY GENERATORS**

In Python 2.4, a relative of list comprehension (see Chapter 5) was introduced: *generator comprehension* (or *generator expressions*). It works in the same way as list comprehension, except that a list isn't constructed (and the "body" isn't looped over immediately). Instead, a generator is returned, allowing you to perform the computation step by step:

```
>>> g = ((i+2)**2 for i in range(2,27))
>>> g.next()
16
```

As you can see, this differs from list comprehension in the use of plain parentheses. In a simple case such as this, I might as well have used a list comprehension. However, if you wish to "wrap" an iterable object (possibly yielding a huge number of values), a list comprehension would void the advantages of iteration by immediately instantiating a list.

A neat bonus is that when using generator comprehension directly inside a pair of existing parentheses, such as in a function call, you don't need to add another pair. In other words, you can write pretty code like this:

```
sum(i**2 for i in range(10))
```

## A Recursive Generator

The generator I designed in the previous section could deal only with lists nested two levels deep, and to do that it used two `for` loops. What if you have a set of lists nested arbitrarily deeply? Perhaps you use them to represent some tree structure, for example. (You can also do that with specific tree classes, but the strategy is the same.) You need a `for` loop for each level of nesting, but because you don't know how many levels there are, you must change your solution to be more flexible. It's time to turn to the magic of recursion:

```
def flatten(nested):
    try:
        for sublist in nested:
            for element in flatten(sublist):
                yield element
    except TypeError:
        yield nested
```

When `flatten` is called, you have two possibilities (as is always the case when dealing with recursion): the *base* case and the *recursive* case. In the base case, the function is told to flatten a single element (for example, a number), in which case the `for` loop raises a `TypeError` (because you're trying to iterate over a number), and the generator simply yields the element.

If you are told to flatten a list (or any iterable), however, you need to do some work. You go through all the sublists (some of which may not really be lists) and call `flatten` on them. Then you yield all the elements of the flattened sublists by using another `for` loop. It may seem slightly magical, but it works:

```
>>> list(flatten([[[1],2],3,4,[5,[6,7]],8]))
[1, 2, 3, 4, 5, 6, 7, 8]
```

There is one problem with this, however. If `nested` is a string-like object (string, Unicode, `UserString`, and so on), it is a sequence and will not raise `TypeError`, yet you do *not* want to iterate over it.

**■Note** There are two main reasons why you shouldn't iterate over string-like objects in the `flatten` function. First, you want to treat string-like objects as atomic values, not as sequences that should be flattened. Second, iterating over them would actually lead to infinite recursion because the first element of a string is another string of length one, and the first element of *that* string is the string itself!

To deal with this, you must add a test at the beginning of the generator. Trying to concatenate the object with a string and seeing if a `TypeError` results is the simplest and fastest way to check whether an object is string-like.[2] Here is the generator with the added test:

```
def flatten(nested):
    try:
        # Don't iterate over string-like objects:
        try: nested + ''
        except TypeError: pass
        else: raise TypeError
        for sublist in nested:
            for element in flatten(sublist):
                yield element
    except TypeError:
        yield nested
```

As you can see, if the expression `nested + ''` raises a `TypeError`, it is ignored; however, if the expression does *not* raise a `TypeError`, the `else` clause of the inner `try` statement raises a `TypeError` of its own. This causes the string-like object to be yielded as is (in the outer `except` clause). Got it?

Here is an example to demonstrate that this version works with strings as well:

```
>>> list(flatten(['foo', ['bar', ['baz']]]))
['foo', 'bar', 'baz']
```

Note that there is no type checking going on here. I don't test whether `nested` *is* a string (which I could do by using `isinstance`), only whether it *behaves* like one (that is, it can be concatenated with a string).

## Generators in General

If you followed the examples so far, you know how to use generators, more or less. You've seen that a generator is a function that contains the keyword `yield`. When it is called, the code in the function body is not executed. Instead, an iterator is returned. Each time a value is requested,

2. Thanks to Alex Martelli for pointing out this idiom and the importance of using it here.

the code in the generator is executed until a yield or a return is encountered. A yield means that a value should be yielded. A return means that the generator should stop executing (without yielding anything more; return can be called without arguments only when used inside a generator).

In other words, generators consist of two separate components: the *generator-function* and the *generator-iterator*. The generator-function is what is defined by the def statement containing a yield. The generator-iterator is what this function returns. In less precise terms, these two entities are often treated as one and collectively called *a generator*.

```
>>> def simple_generator():
        yield 1
...
>>> simple_generator
<function simple_generator at 153b44>
>>> simple_generator()
<generator object at 1510b0>
```

The iterator returned by the generator-function can be used just like any other iterator.

## Generator Methods

A relatively new feature of generators (added in Python 2.5) is the ability to supply generators with values after they have started running. This takes the form of a communications channel between the generator and the "outside world," with the following two end points:

- The outside world has access to a method on the generator called send, which works just like next, except that it takes a single argument (the "message" to send—an arbitrary object).
- Inside the suspended generator, yield may now be used as an *expression*, rather than a *statement*. In other words, when the generator is resumed, yield returns a value—the value sent from the outside through send. If next was used, yield returns None.

Note that using send (rather than next) makes sense only after the generator has been suspended (that is, after it has hit the first yield). If you need to give some information to the generator before that, you can simply use the parameters of the generator-function.

■ **Tip** If you *really* want to use send on a newly started generator, you can use it with None as its parameter.

Here's a rather silly example that illustrates the mechanism:

```
def repeater(value):
    while True:
        new = (yield value)
        if new is not None: value = new
```

Here's an example of its use:

```
r = repeater(42)
r.next()
42
r.send("Hello, world!")
"Hello, world!"
```

Note the use of parentheses around the `yield` expression. While not strictly necessary in some cases, it is probably better to be safe than sorry, and simply always enclose `yield` expressions in parentheses if you are using the return value in some way.

Generators also have two other methods (in Python 2.5 and later):

- The `throw` method (called with an exception type, an optional value and traceback object) is used to raise an exception inside the generator (at the `yield` expression).
- The `close` method (called with no arguments) is used to stop the generator.

The `close` method (which is also called by the Python garbage collector, when needed) is also based on exceptions. It raises the `GeneratorExit` exception at the yield point, so if you want to have some cleanup code in your generator, you can wrap your `yield` in a `try/finally` statement. If you wish, you can also catch the `GeneratorExit` exception, but then you must reraise it (possibly after cleaning up a bit), raise another exception, or simply return. Trying to yield a value from a generator after `close` has been called on it will result in a `RuntimeError`.

■ **Tip** For more information about generator methods, and how these actually turn generators into simple *coroutines*, see PEP 342 (`http://www.python.org/dev/peps/pep-0342/`).

## Simulating Generators

If you need to use an older version of Python, generators aren't available. What follows is a simple recipe for simulating them with normal functions.

Starting with the code for the generator, begin by inserting the following line at the beginning of the function body:

```
result = []
```

If the code already uses the name `result`, you should come up with another. (Using a more descriptive name may be a good idea anyway.) Then replace all lines of this form:

```
yield some_expression
```

with this:

```
result.append(some_expression)
```

Finally, at the end of the function, add this line:

```
return result
```

Although this may not work with all generators, it works with most. (For example, it fails with infinite generators, which of course can't stuff their values into a list.)

Here is the `flatten` generator rewritten as a plain function:

```
def flatten(nested):
    result = []
    try:
        # Don't iterate over string-like objects:
        try: nested + ''
        except TypeError: pass
        else: raise TypeError
        for sublist in nested:
            for element in flatten(sublist):
                result.append(element)
    except TypeError:
        result.append(nested)
    return result
```

## The Eight Queens

Now that you've learned about all this magic, it's time to put it to work. In this section, you see how to use generators to solve a classic programming problem.

### Generators and Backtracking

Generators are ideal for complex recursive algorithms that gradually build a result. Without generators, these algorithms usually require you to pass a half-built solution around as an extra parameter so that the recursive calls can build on it. With generators, all the recursive calls need to do is `yield` their part. That is what I did with the preceding recursive version of `flatten`, and you can use the exact same strategy to traverse graphs and tree structures.

In some applications, however, you don't get the answer right away; you need to try several alternatives, and you need to do that on *every* level in your recursion. To draw a parallel from real life, imagine that you have an important meeting to attend. You're not sure where it is, but you have two doors in front of you, and the meeting room has to be behind one of them. You choose the left and step through. There, you face another two doors. You choose the left, but it turns out to be wrong. So you *backtrack*, and choose the right door, which also turns out to be wrong (excuse the pun). So, you backtrack again, to the point where you started, ready to try the right door there.

### GRAPHS AND TREES

If you have never heard of graphs and trees before, you should learn about them as soon as possible, because they are very important concepts in programming and computer science. To find out more, you should probably get a book about computer science, discrete mathematics, data structures, or algorithms. For some concise definitions, you can check out the following web pages:

- `http://mathworld.wolfram.com/Graph.html`
- `http://mathworld.wolfram.com/Tree.html`
- `http://www.nist.gov/dads/HTML/tree.html`
- `http://www.nist.gov/dads/HTML/graph.html`

A quick web search or some browsing in Wikipedia (`http://wikipedia.org`) will turn up a lot of material.

This strategy of backtracking is useful for solving problems that require you to try every combination until you find a solution. Such problems are solved like this:

```
# Pseudocode
for each possibility at level 1:
    for each possibility at level 2:
        ...
            for each possibility at level n:
                is it viable?
```

To implement this directly with `for` loops, you need to know how many levels you'll encounter. If that is not possible, you use recursion.

## The Problem

This is a much loved computer science puzzle: you have a chessboard and eight queen pieces to place on it. The only requirement is that none of the queens threatens any of the others; that is, you must place them so that no two queens can capture each other. How do you do this? Where should the queens be placed?

This is a typical backtracking problem: you try one position for the first queen (in the first row), advance to the second, and so on. If you find that you are unable to place a queen, you backtrack to the previous one and try another position. Finally, you either exhaust all possibilities or find a solution.

In the problem as stated, you are provided with information that there will be only eight queens, but let's assume that there can be any number of queens. (This is more similar to real-world backtracking problems.) How do you solve that? If you want to try to solve it yourself, you should stop reading now, because I'm about to give you the solution.

---

■ **Note** You can find much more efficient solutions for this problem. If you want more details, a web search should turn up a wealth of information. A brief history of various solutions may be found at `http://www.cit.gu.edu.au/~sosic/nqueens.html`.

---

## State Representation

To represent a possible solution (or part of it), you can simply use a tuple (or a list, for that matter). Each element of the tuple indicates the position (that is, column) of the queen of the corresponding row. So if `state[0] == 3`, you know that the queen in row one is positioned in column four (we are counting from zero, remember?). When working at one level of recursion (one specific row), you know only which positions the queens above have, so you may have a state tuple whose length is less than eight (or whatever the number of queens is).

---

■ **Note** I could well have used a list instead of a tuple to represent the state. It's mostly a matter of taste in this case. In general, if the sequence is small and static, tuples may be a good choice.

---

## Finding Conflicts

Let's start by doing some simple abstraction. To find a configuration in which there are no conflicts (where no queen may capture another), you first must define what a conflict is. And why not define it as a function while you're at it?

The `conflict` function is given the positions of the queens *so far* (in the form of a state tuple) and determines if a position for the *next* queen generates any new conflicts:

```
def conflict(state, nextX):
    nextY = len(state)
    for i in range(nextY):
        if abs(state[i]-nextX) in (0, nextY-i):
            return True
    return False
```

The `nextX` parameter is the suggested horizontal position (x coordinate, or column) of the next queen, and `nextY` is the vertical position (y coordinate, or row) of the next queen. This function does a simple check for each of the previous queens. If the next queen has the same x coordinate, or is on the same diagonal as `(nextX, nextY)`, a conflict has occurred, and `True` is returned. If no such conflicts arise, `False` is returned. The tricky part is the following expression:

```
abs(state[i]-nextX) in (0, nextY-i)
```

It is true if the horizontal distance between the next queen and the previous one under consideration is either zero (same column) or equal to the vertical distance (on a diagonal). Otherwise, it is false.

## The Base Case

The Eight Queens problem can be a bit tricky to implement, but with generators it isn't so bad. If you aren't used to recursion, I wouldn't expect you to come up with this solution by yourself, though. Note also that this solution isn't particularly efficient, so with a very large number of queens, it might be a bit slow.

Let's begin with the base case: the last queen. What would you want her to do? Let's say you want to find all possible solutions. In that case, you would expect her to produce (generate) all the positions she could occupy (possibly none) given the positions of the others. You can sketch this out directly:

```
def queens(num, state):
    if len(state) == num-1:
        for pos in range(num):
            if not conflict(state, pos):
                yield pos
```

In human-speak, this means, "If all queens but one have been placed, go through all possible positions for the last one, and return the positions that don't give rise to any conflicts." The `num` parameter is the number of queens in total, and the `state` parameter is the tuple of positions for the previous queens. For example, let's say you have four queens, and that the first three have been given the positions 1, 3, and 0, respectively, as shown in Figure 9-1. (Pay no attention to the white queen at this point.)

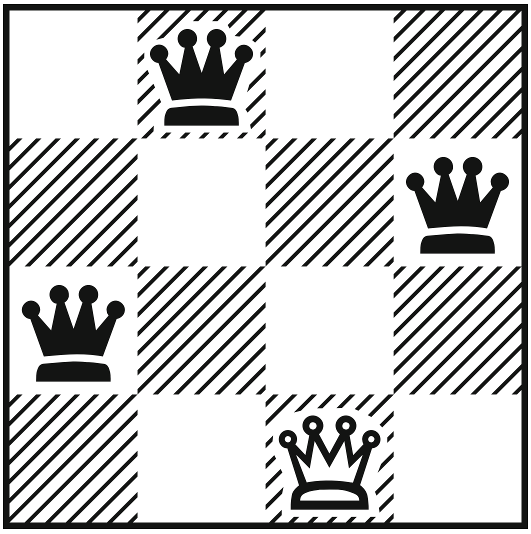

**Figure 9-1.** *Placing four queens on a 4 × 4 board*

As you can see in the figure, each queen gets a (horizontal) row, and the queens' positions are numbered across the top (beginning with zero, as is normal in Python):

```
>>> list(queens(4, (1,3,0)))
[2]
```

It works like a charm. Using `list` simply forces the generator to yield all of its values. In this case, only one position qualifies. The white queen has been put in this position in Figure 9-1. (Note that color has no special significance and is not part of the program.)

## The Recursive Case

Now let's turn to the recursive part of the solution. When you have your base case covered, the recursive case may correctly assume (by induction) that all results from lower levels (the queens with higher numbers) are correct. So what you need to do is add an `else` clause to the `if` statement in the previous implementation of the `queens` function.

What results do you expect from the recursive call? You want the positions of all the lower queens, right? Let's say they are returned as a tuple. In that case, you probably need to change your base case to return a tuple as well (of length one)—but I get to that later.

So, you're supplied with one tuple of positions from "above," and for each legal position of the current queen, you are supplied with a tuple of positions from "below." All you need to do to keep things flowing is to yield the result from below with your own position added to the front:

```
    ...
    else:
        for pos in range(num):
            if not conflict(state, pos):
                for result in queens(num, state + (pos,)):
                    yield (pos,) + result
```

The `for pos` and `if not conflict` parts of this are identical to what you had before, so you can rewrite this a bit to simplify the code. Let's add some default arguments as well:

```
def queens(num=8, state=()):
    for pos in range(num):
        if not conflict(state, pos):
            if len(state) == num-1:
                yield (pos,)
            else:
                for result in queens(num, state + (pos,)):
                    yield (pos,) + result
```

If you find the code hard to understand, you might find it helpful to formulate what it does in your own words. (And you do remember that the comma in `(pos,)` is necessary to make it a tuple, and not simply a parenthesized value, right?)

The queens generator gives you all the solutions (that is, all the legal ways of placing the queens):

```
>>> list(queens(3))
[]
>>> list(queens(4))
[(1, 3, 0, 2), (2, 0, 3, 1)]
>>> for solution in queens(8):
...     print solution
...
(0, 4, 7, 5, 2, 6, 1, 3)
(0, 5, 7, 2, 6, 3, 1, 4)
...
(7, 2, 0, 5, 1, 4, 6, 3)
(7, 3, 0, 2, 5, 1, 6, 4)
>>>
```

If you run queens with eight queens, you see a lot of solutions flashing by. Let's find out how many:

```
>>> len(list(queens(8)))
92
```

## Wrapping It Up

Before leaving the queens, let's make the output a bit more understandable. Clear output is always a good thing because it makes it easier to spot bugs, among other things.

```
def prettyprint(solution):
    def line(pos, length=len(solution)):
        return '. ' * (pos) + 'X ' + '. ' * (length-pos-1)
    for pos in solution:
        print line(pos)
```

Note that I've made a little helper function inside prettyprint. I put it there because I assumed I wouldn't need it anywhere outside. In the following, I print out a random solution to satisfy myself that it is correct:

```
>>> import random
>>> prettyprint(random.choice(list(queens(8))))
. . . . . X . .
. X . . . . . .
. . . . . . X .
X . . . . . . .
. . . X . . . .
. . . . . . . X
. . . . X . . .
. . X . . . . .
```

This “drawing” corresponds to the diagram in Figure 9-2. Fun to play with Python, isn’t it?

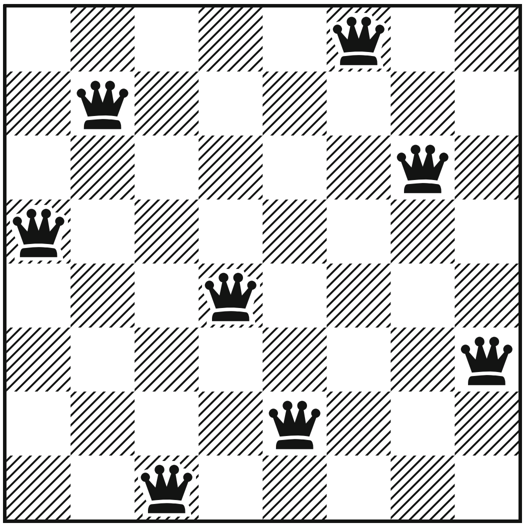

**Figure 9-2.** *One of many possible solutions to the Eight Queens problem*

## A Quick Summary

You’ve seen a lot of magic here. Let’s take stock:

**New-style vs. old-style classes**: The way classes work in Python is changing. Recent (pre-3.0) versions of Python have two sorts of classes, with the old-style ones quickly going out of fashion. The new-style classes were introduced in version 2.2, and they provide several extra features (for example, they work with `super` and `property`, while old-style classes do not). To create a new-style class, you must subclass `object`, either directly or indirectly, or set the `__metaclass__` property.

**Magic methods**: Several special methods (with names beginning and ending with double underscores) exist in Python. These methods differ quite a bit in function, but most of them are called automatically by Python under certain circumstances. (For example, `__init__` is called after object creation.)

**Constructors**: These are common to many object-oriented languages, and you’ll probably implement one for almost every class you write. Constructors are named `__init__` and are automatically called immediately after an object is created.

**Overriding**: A class can override methods (or any other attributes) defined in its superclasses simply by implementing the methods. If the new method needs to call the overridden version, it can either call the unbound version from the superclass directly (old-style classes) or use the `super` function (new-style classes).

**Sequences and mappings**: Creating a sequence or mapping of your own requires implementing all the methods of the sequence and mapping protocols, including such magic methods as `__getitem__` and `__setitem__`. By subclassing `list` (or `UserList`) and `dict` (or `UserDict`), you can save a lot of work.

**Iterators**: An *iterator* is simply an object that has a `next` method. Iterators can be used to iterate over a set of values. When there are no more values, the `next` method should raise a `StopIteration` exception. *Iterable* objects have an `__iter__` method, which returns an iterator, and can be used in `for` loops, just like sequences. Often, an iterator is also iterable; that is, it has an `__iter__` method that returns the iterator itself.

**Generators**: A *generator-function* (or method) is a function (or method) that contains the keyword `yield`. When called, the generator-function returns a *generator*, which is a special type of iterator. You can interact with an active generator from the outside by using the methods `send`, `throw`, and `close`.

**Eight Queens**: The Eight Queens problem is well known in computer science and lends itself easily to implementation with generators. The goal is to position eight queens on a chessboard so that none of the queens is in a position from which she can attack any of the others.

## New Functions in This Chapter

| Function | Description |
|---|---|
| `iter(obj)` | Extracts an iterator from an iterable object |
| `property(fget, fset, fdel, doc)` | Returns a property; all arguments are optional |
| `super(class, obj)` | Returns a bound instance of `class`'s superclass |

Note that `iter` and `super` may be called with other parameters than those described here. For more information, see the standard Python documentation (`http://python.org/doc`).

## What Now?

Now you know most of the Python language. So why are there still so many chapters left? Well, there is still a *lot* to learn, much of it about how Python can connect to the external world in various ways. And then we have testing, extending, packaging, and the projects, so we're not done yet—not by far.