7

Classes and Object-Oriented Programming

Classes are the mechanism used to create new kinds of objects. This chapter covers the details of classes, but is not intended to be an in-depth reference on object-oriented programming and design. It's assumed that the reader has some prior experience with data structures and object-oriented programming in other languages such as C or Java. (Chapter 3, "Types and Objects," contains additional information about the terminology and internal implementation of objects.)

The `class` Statement

A *class* defines a set of attributes that are associated with, and shared by, a collection of objects known as *instances*. A class is most commonly a collection of functions (known as *methods*), variables (which are known as *class variables*), and computed attributes (which are known as *properties*).

A class is defined using the `class` statement. The body of a class contains a series of statements that execute during class definition. Here's an example:

```
class Account(object):
     num_accounts = 0
     def __init__(self,name,balance):
         self.name = name
         self.balance = balance
         Account.num_accounts += 1
     def __del__(self):
         Account.num_accounts -= 1
     def deposit(self,amt):
         self.balance = self.balance + amt
     def withdraw(self,amt):
         self.balance = self.balance - amt
     def inquiry(self):
         return self.balance
```

The values created during the execution of the class body are placed into a class object that serves as a namespace much like a module. For example, the members of the `Account` class are accessed as follows:

```
Account.num_accounts
Account.__init__
Account.__del__
Account.deposit
Account.withdraw
Account.inquiry
```

It's important to note that a `class` statement by itself doesn't create any instances of the class (for example, no accounts are actually created in the preceding example). Rather, a class merely sets up the attributes that will be common to all the instances that will be created later. In this sense, you might think of it as a blueprint.

The functions defined inside a class are known as *instance methods*. An instance method is a function that operates on an instance of the class, which is passed as the first argument. By convention, this argument is called `self`, although any legal identifier name can be used. In the preceding example, `deposit()`, `withdraw()`, and `inquiry()` are examples of instance methods.

Class variables such as `num_accounts` are values that are shared among all instances of a class (that is, they're not individually assigned to each instance). In this case, it's a variable that's keeping track of how many `Account` instances are in existence.

Class Instances

Instances of a class are created by calling a class object as a function. This creates a new instance that is then passed to the `__init__()` method of the class. The arguments to `__init__()` consist of the newly created instance `self` along with the arguments supplied when calling the class object. For example:

```
# Create a few accounts
a = Account("Guido", 1000.00)  # Invokes Account.__init__(a,"Guido",1000.00)
b = Account("Bill", 10.00)
```

Inside `__init__()`, attributes are saved in the instance by assigning to `self`. For example, `self.name = name` is saving a `name` attribute in the instance. Once the newly created instance has been returned to the user, these attributes as well as attributes of the class are accessed using the dot (.) operator as follows:

```
a.deposit(100.00)        # Calls Account.deposit(a,100.00)
b.withdraw(50.00)        # Calls Account.withdraw(b,50.00)
name = a.name            # Get account name
```

The dot (.) operator is responsible for attribute binding. When you access an attribute, the resulting value may come from several different places. For example, `a.name` in the previous example returns the `name` attribute of the instance `a`. However, `a.deposit` returns the `deposit` attribute (a method) of the `Account` class. When you access an attribute, the instance is checked first and if nothing is known, the search moves to the instance's class instead. This is the underlying mechanism by which a class shares its attributes with all of its instances.

Scoping Rules

Although classes define a namespace, classes do not create a scope for names used inside the bodies of methods. Therefore, when you're implementing a class, references to attributes and methods must be fully qualified. For example, in methods you always reference attributes of the instance through `self`. Thus, in the example you use `self.balance`, not `balance`. This also applies if you want to call a method from another method, as shown in the following example:

```
class Foo(object):
    def bar(self):
        print("bar!")
    def spam(self):
        bar(self)     # Incorrect! 'bar' generates a NameError
        self.bar()    # This works
        Foo.bar(self) # This also works
```

The lack of scoping in classes is one area where Python differs from C++ or Java. If you have used those languages, the `self` parameter in Python is the same as the `this` pointer. The explicit use of `self` is required because Python does not provide a means to explicitly declare variables (that is, a declaration such as `int x` or `float y` in C). Without this, there is no way to know whether an assignment to a variable in a method is supposed to be a local variable or if it's supposed to be saved as an instance attribute. The explicit use of `self` fixes this—all values stored on `self` are part of the instance and all other assignments are just local variables.

Inheritance

Inheritance is a mechanism for creating a new class that specializes or modifies the behavior of an existing class. The original class is called a *base class* or a *superclass*. The new class is called a *derived class* or a *subclass*. When a class is created via inheritance, it "inherits" the attributes defined by its base classes. However, a derived class may redefine any of these attributes and add new attributes of its own.

Inheritance is specified with a comma-separated list of base-class names in the `class` statement. If there is no logical base class, a class inherits from `object`, as has been shown in prior examples. `object` is a class which is the root of all Python objects and which provides the default implementation of some common methods such as `__str__()`, which creates a string for use in printing.

Inheritance is often used to redefine the behavior of existing methods. As an example, here's a specialized version of `Account` that redefines the `inquiry()` method to periodically overstate the current balance with the hope that someone not paying close attention will overdraw his account and incur a big penalty when making a payment on their subprime mortgage:

```
import random
class EvilAccount(Account):
    def inquiry(self):
        if random.randint(0,4) == 1:
           return self.balance * 1.10   # Note: Patent pending idea
        else:
           return self.balance

c = EvilAccount("George", 1000.00)
c.deposit(10.0)         # Calls Account.deposit(c,10.0)
available = c.inquiry() # Calls EvilAccount.inquiry(c)
```

In this example, instances of `EvilAccount` are identical to instances of `Account` except for the redefined `inquiry()` method.

Inheritance is implemented with only a slight enhancement of the dot (.) operator. Specifically, if the search for an attribute doesn't find a match in the instance or the instance's class, the search moves on to the base class. This process continues until there are no more base classes to search. In the previous example, this explains why `c.deposit()` calls the implementation of `deposit()` defined in the `Account` class.

A subclass can add new attributes to the instances by defining its own version of `__init__()`. For example, this version of `EvilAccount` adds a new attribute `evilfactor`:

```
class EvilAccount(Account):
    def __init__(self,name,balance,evilfactor):
        Account.__init__(self,name,balance)    # Initialize Account
        self.evilfactor = evilfactor
    def inquiry(self):
        if random.randint(0,4) == 1:
           return self.balance * self.evilfactor
        else:
           return self.balance
```

When a derived class defines `__init__()`, the `__init__()` methods of base classes are not automatically invoked. Therefore, it's up to a derived class to perform the proper initialization of the base classes by calling their `__init__()` methods. In the previous example, this is shown in the statement that calls `Account.__init__()`. If a base class does not define `__init__()`, this step can be omitted. If you don't know whether the base class defines `__init__()`, it is always safe to call it without any arguments because there is always a default implementation that simply does nothing.

Occasionally, a derived class will reimplement a method but also want to call the original implementation. To do this, a method can explicitly call the original method in the base class, passing the instance `self` as the first parameter as shown here:

```
class MoreEvilAccount(EvilAccount):
    def deposit(self,amount):
        self.withdraw(5.00)             # Subtract the "convenience" fee
        EvilAccount.deposit(self,amount) # Now, make deposit
```

A subtlety in this example is that the class `EvilAccount` doesn't actually implement the `deposit()` method. Instead, it is implemented in the `Account` class. Although this code works, it might be confusing to someone reading the code (e.g., was `EvilAccount` supposed to implement `deposit()`?). Therefore, an alternative solution is to use the `super()` function as follows:

```
class MoreEvilAccount(EvilAccount):
    def deposit(self,amount):
        self.withdraw(5.00)                       # Subtract convenience fee
        super(MoreEvilAccount,self).deposit(amount) # Now, make deposit
```

`super(`*`cls`*`, `*`instance`*`)` returns a special object that lets you perform attribute lookups on the base classes. If you use this, Python will search for an attribute using the normal search rules that would have been used on the base classes. This frees you from hard-coding the exact location of a method and more clearly states your intentions (that is, you want to call the previous implementation without regard for which base class defines it). Unfortunately, the syntax of `super()` leaves much to be desired. If you are using Python 3, you can use the simplified statement `super().deposit(amount)` to carry out the calculation shown in the example. In Python 2, however, you have to use the more verbose version.

Python supports multiple inheritance. This is specified by having a class list multiple base classes. For example, here are a collection of classes:

```
class DepositCharge(object):
    fee = 5.00
    def deposit_fee(self):
        self.withdraw(self.fee)

class WithdrawCharge(object):
    fee = 2.50
    def withdraw_fee(self):
        self.withdraw(self.fee)

# Class using multiple inheritance
class MostEvilAccount(EvilAccount, DepositCharge, WithdrawCharge):
    def deposit(self,amt):
        self.deposit_fee()
        super(MostEvilAccount,self).deposit(amt)
    def withdraw(self,amt):
        self.withdraw_fee()
        super(MostEvilAcount,self).withdraw(amt)
```

When multiple inheritance is used, attribute resolution becomes considerably more complicated because there are many possible search paths that could be used to bind attributes. To illustrate the possible complexity, consider the following statements:

```
d = MostEvilAccount("Dave",500.00,1.10)
d.deposit_fee()    # Calls DepositCharge.deposit_fee().   Fee is 5.00
d.withdraw_fee()   # Calls WithdrawCharge.withdraw_fee(). Fee is 5.00 ??
```

In this example, methods such as `deposit_fee()` and `withdraw_fee()` are uniquely named and found in their respective base classes. However, the `withdraw_fee()` function doesn't seem to work right because it doesn't actually use the value of `fee` that was initialized in its own class. What has happened is that the attribute `fee` is a class variable defined in two different base classes. One of those values is used, but which one? (Hint: it's `DepositCharge.fee`.)

To find attributes with multiple inheritance, all base classes are ordered in a list from the "most specialized" class to the "least specialized" class. Then, when searching for an attribute, this list is searched in order until the first definition of the attribute is found. In the example, the class `EvilAccount` is more specialized than `Account` because it inherits from `Account`. Similarly, within `MostEvilAccount`, `DepositCharge` is considered to be more specialized than `WithdrawCharge` because it is listed first in the list of base classes. For any given class, the ordering of base classes can be viewed by printing its `__mro__` attribute. Here's an example:

```
>>> MostEvilAccount.__mro__
(<class '__main__.MostEvilAccount'>,
 <class '__main__.EvilAccount'>,
 <class '__main__.Account'>,
 <class '__main__.DepositCharge'>,
 <class '__main__.WithdrawCharge'>,
 <type 'object'>)
>>>
```

In most cases, this list is based on rules that "make sense." That is, a derived class is always checked before its base classes and if a class has more than one parent, the parents are always checked in the same order as listed in the class definition. However, the precise ordering of base classes is actually quite complex and not based on any sort of "simple" algorithm such as depth-first or breadth-first search. Instead, the ordering is determined according to the C3 linearization algorithm, which is described in the paper "A Monotonic Superclass Linearization for Dylan" (K. Barrett, et al, presented at

OOPSLA'96). A subtle aspect of this algorithm is that certain class hierarchies will be rejected by Python with a `TypeError`. Here's an example:

```
class X(object): pass
class Y(X): pass
class Z(X,Y): pass  # TypeError.
                    # Can't create consistent method resolution order__
```

In this case, the method resolution algorithm rejects class `Z` because it can't determine an ordering of the base classes that makes sense. For example, the class `X` appears before class `Y` in the inheritance list, so it must be checked first. However, class `Y` is more specialized because it inherits from `X`. Therefore, if `X` is checked first, it would not be possible to resolve specialized methods in `Y`. In practice, these issues should rarely arise—and if they do, it usually indicates a more serious design problem with a program.

As a general rule, multiple inheritance is something best avoided in most programs. However, it is sometimes used to define what are known as *mixin* classes. A mixin class typically defines a set of methods that are meant to be "mixed in" to other classes in order to add extra functionality (almost like a macro). Typically, the methods in a mixin will assume that other methods are present and will build upon them. The `DepositCharge` and `WithdrawCharge` classes in the earlier example illustrate this. These classes add new methods such as `deposit_fee()` to classes that include them as one of the base classes. However, you would never instantiate `DepositCharge` by itself. In fact, if you did, it wouldn't create an instance that could be used for anything useful (that is, the one defined method wouldn't even execute correctly).

Just as a final note, if you wanted to fix the problematic references to `fee` in this example, the implementation of `deposit_fee()` and `withdraw_fee()` should be changed to refer to the attribute directly using the class name instead of `self` (for example, `DepositChange.fee`).

Polymorphism Dynamic Binding and Duck Typing

Dynamic binding (also sometimes referred to as *polymorphism* when used in the context of inheritance) is the capability to use an instance without regard for its type. It is handled entirely through the attribute lookup process described for inheritance in the preceding section. Whenever an attribute is accessed as *`obj`*`.`*`attr`*, *`attr`* is located by searching within the instance itself, the instance's class definition, and then base classes, in that order. The first match found is returned.

A critical aspect of this binding process is that it is independent of what kind of object *`obj`* is. Thus, if you make a lookup such as *`obj`*`.name`, it will work on any *`obj`* that happens to have a `name` attribute. This behavior is sometimes referred to as *duck typing* in reference to the adage "if it looks like, quacks like, and walks like a duck, then it's a duck."

Python programmers often write programs that rely on this behavior. For example, if you want to make a customized version of an existing object, you can either inherit from it or you can simply create a completely new object that looks and acts like it but is otherwise unrelated. This latter approach is often used to maintain a loose coupling of program components. For example, code may be written to work with any kind of object whatsoever as long as it has a certain set of methods. One of the most common examples is with various "file-like" objects defined in the standard library. Although these objects work like files, they don't inherit from the built-in file object.

Static Methods and Class Methods

In a class definition, all functions are assumed to operate on an instance, which is always passed as the first parameter `self`. However, there are two other common kinds of methods that can be defined.

A *static method* is an ordinary function that just happens to live in the namespace defined by a class. It does not operate on any kind of instance. To define a static method, use the `@staticmethod` decorator as shown here:

```
class Foo(object):
     @staticmethod
     def add(x,y):
         return x + y
```

To call a static method, you just prefix it by the class name. You do not pass it any additional information. For example:

```
x = Foo.add(3,4)       # x  = 7
```

A common use of static methods is in writing classes where you might have many different ways to create new instances. Because there can only be one `__init__()` function, alternative creation functions are often defined as shown here:

```
class Date(object):
    def __init__(self,year,month,day):
        self.year = year
        self.month = month
        self.day = day
    @staticmethod
    def now():
         t = time.localtime()
         return Date(t.tm_year, t.tm_mon, t.tm_day)
    @staticmethod
    def tomorrow():
         t = time.localtime(time.time()+86400)
         return Date(t.tm_year, t.tm_mon, t.tm_day)

# Example of creating some dates
a = Date(1967, 4, 9)
b = Date.now()       # Calls static method now()
c = Date.tomorrow()  # Calls static method tomorrow()
```

Class methods are methods that operate on the class itself as an object. Defined using the `@classmethod` decorator, a class method is different than an instance method in that the class is passed as the first argument which is named `cls` by convention. For example:

```
class Times(object):
    factor = 1
    @classmethod
    def mul(cls,x):
        return cls.factor*x

class TwoTimes(Times):
    factor = 2

x = TwoTimes.mul(4)      # Calls Times.mul(TwoTimes, 4) -> 8
```

In this example, notice how the class `TwoTimes` is passed to `mul()` as an object. Although this example is esoteric, there are practical, but subtle, uses of class methods. As an example, suppose that you defined a class that inherited from the `Date` class shown previously and customized it slightly:

```
class EuroDate(Date):
    # Modify string conversion to use European dates
    def __str__(self):
        return "%02d/%02d/%4d" % (self.day, self.month, self.year)
```

Because the class inherits from `Date`, it has all of the same features. However, the `now()` and `tomorrow()` methods are slightly broken. For example, if someone calls `EuroDate.now()`, a `Date` object is returned instead of a `EuroDate` object. A class method can fix this:

```
class Date(object):
    ...
    @classmethod
    def now(cls):
         t = time.localtime()
         # Create an object of the appropriate type
         return cls(t.tm_year, t.tm_month, t.tm_day)

class EuroDate(Date):
    ...

a = Date.now()       # Calls Date.now(Date) and returns a Date
b = EuroDate.now()   # Calls Date.now(EuroDate) and returns a EuroDate
```

One caution about static and class methods is that Python does not manage these methods in a separate namespace than the instance methods. As a result, they can be invoked on an instance. For example:

```
a = Date(1967,4,9)
b = d.now()          # Calls Date.now(Date)
```

This is potentially quite confusing because a call to `d.now()` doesn't really have anything to do with the instance `d`. This behavior is one area where the Python object system differs from that found in other OO languages such as Smalltalk and Ruby. In those languages, class methods are strictly separate from instance methods.

Properties

Normally, when you access an attribute of an instance or a class, the associated value that is stored is returned. A *property* is a special kind of attribute that computes its value when accessed. Here is a simple example:

```
class Circle(object):
     def __init__(self,radius):
         self.radius = radius
     # Some additional properties of Circles
     @property
     def area(self):
         return math.pi*self.radius**2
     @property
     def perimeter(self):
         return 2*math.pi*self.radius
```

The resulting `Circle` object behaves as follows:

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area
50.26548245743669
>>> c.perimeter
25.132741228718345
>>> c.area = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

In this example, `Circle` instances have an instance variable `c.radius` that is stored. `c.area` and `c.perimeter` are simply computed from that value. The `@property` decorator makes it possible for the method that follows to be accessed as a simple attribute, without the extra `()` that you would normally have to add to call the method. To the user of the object, there is no obvious indication that an attribute is being computed other than the fact that an error message is generated if an attempt is made to redefine the attribute (as shown in the `AttributeError` exception above).

Using properties in this way is related to something known as the *Uniform Access Principle*. Essentially, if you're defining a class, it is always a good idea to make the programming interface to it as uniform as possible. Without properties, certain attributes of an object would be accessed as a simple attribute such as `c.radius` whereas other attributes would be accessed as methods such as `c.area()`. Keeping track of when to add the extra `()` adds unnecessary confusion. A property can fix this.

Python programmers don't often realize that methods themselves are implicitly handled as a kind of property. Consider this class:

```
class Foo(object):
     def __init__(self,name):
         self.name = name
     def spam(self,x):
         print("%s, %s" % (self.name, x)
```

When a user creates an instance such as `f = Foo("Guido")` and then accesses `f.spam`, the original function object `spam` is not returned. Instead, you get something known as a *bound method*, which is an object that represents the method call that will execute when the `()` operator is invoked on it. A bound method is like a partially evaluated function where the `self` parameter has already been filled in, but the additional arguments still need to be supplied by you when you call it using `()`. The creation of this bound method object is silently handled through a property function that executes behind the scenes. When you define static and class methods using `@staticmethod` and `@classmethod`, you are actually specifying the use of a different property function that will handle the access to those methods in a different way. For example, `@staticmethod` simply returns the method function back "as is" without any special wrapping or processing.

Properties can also intercept operations to set and delete an attribute. This is done by attaching additional setter and deleter methods to a property. Here is an example:

```
class Foo(object):
    def __init__(self,name):
        self.__name = name
    @property
    def name(self):
        return self.__name
    @name.setter
    def name(self,value):
        if not isinstance(value,str):
             raise TypeError("Must be a string!")
        self.__name = value
    @name.deleter
    def name(self):
        raise TypeError("Can't delete name")

f = Foo("Guido")
n = f.name         # calls f.name() - get function
f.name = "Monty"   # calls setter name(f,"Monty")
f.name = 45        # calls setter name(f,45) -> TypeError
del f.name         # Calls deleter name(f) -> TypeError
```

In this example, the attribute `name` is first defined as a read-only property using the `@property` decorator and associated method. The `@name.setter` and `@name.deleter` decorators that follow are associating additional methods with the set and deletion operations on the `name` attribute. The names of these methods must exactly match the name of the original property. In these methods, notice that the actual value of the name is stored in an attribute `__name`. The name of the stored attribute does not have to follow any convention, but it has to be different than the property in order to distinguish it from the name of the property itself.

In older code, you will often see properties defined using the `property(`*`getf`*`=None,` *`setf`*`=None,` *`delf`*`=None,` *`doc`*`=None)` function with a set of uniquely named methods for carrying out each operation. For example:

```
class Foo(object):
    def getname(self):
        return self.__name
    def setname(self,value):
        if not isinstance(value,str):
             raise TypeError("Must be a string!")
        self.__name = value
    def delname(self):
        raise TypeError("Can't delete name")
    name = property(getname,setname,delname)
```

This older approach is still supported, but the decorator version tends to lead to classes that are a little more polished. For example, if you use decorators, the `get`, `set`, and `delete` functions aren't also visible as methods.

Descriptors

With properties, access to an attribute is controlled by a series of user-defined `get`, `set`, and `delete` functions. This sort of attribute control can be further generalized through the use of a *descriptor object*. A descriptor is simply an object that represents the value of an attribute. By implementing one or more of the special methods `__get__()`, `__set__()`, and `__delete__()`, it can hook into the attribute access mechanism and can customize those operations. Here is an example:

```
class TypedProperty(object):
    def __init__(self,name,type,default=None):
        self.name = "_" + name
        self.type = type
self.default = default if default else type()
    def __get__(self,instance,cls):
        return getattr(instance,self.name,self.default)
    def __set__(self,instance,value):
        if not isinstance(value,self.type):
            raise TypeError("Must be a %s" % self.type)
        setattr(instance,self.name,value)
    def __delete__(self,instance):
        raise AttributeError("Can't delete attribute")

class Foo(object):
    name = TypedProperty("name",str)
    num  = TypedProperty("num",int,42)
```

In this example, the class `TypedProperty` defines a descriptor where type checking is performed when the attribute is assigned and an error is produced if an attempt is made to delete the attribute. For example:

```
f = Foo()
a = f.name           # Implicitly calls Foo.name.__get__(f,Foo)
f.name = "Guido"     # Calls Foo.name.__set__(f,"Guido")
del f.name           # Calls Foo.name.__delete__(f)
```

Descriptors can only be instantiated at the class level. It is not legal to create descriptors on a per-instance basis by creating descriptor objects inside `__init__()` and other methods. Also, the attribute name used by the class to hold a descriptor takes precedence over attributes stored on instances. In the previous example, this is why the descriptor object takes a name parameter and why the name is changed slightly by inserting a leading underscore. In order for the descriptor to store a value on the instance, it has to pick a name that is different than that being used by the descriptor itself.

Data Encapsulation and Private Attributes

By default, all attributes and methods of a class are "public." This means that they are all accessible without any restrictions. It also implies that everything defined in a base class is inherited and accessible within a derived class. This behavior is often undesirable in object-oriented applications because it exposes the internal implementation of an object and can lead to namespace conflicts between objects defined in a derived class and those defined in a base class.

To fix this problem, all names in a class that start with a double underscore, such as `__Foo`, are automatically mangled to form a new name of the form `_`*`Classname`*`__Foo`. This effectively provides a way for a class to have private attributes and methods because private names used in a derived class won't collide with the same private names used in a base class. Here's an example:

```
class A(object):
    def __init__(self):
        self.__X = 3         # Mangled to self._A__X
    def __spam(self):        # Mangled to _A__spam()
        pass
    def bar(self):
        self.__spam()        # Only calls A.__spam()
```

```
class B(A):
    def __init__(self):
        A.__init__(self)
        self.__X = 37       # Mangled to self._B__X
    def __spam(self):       # Mangled to _B__spam()
        pass
```

Although this scheme provides the illusion of data hiding, there's no strict mechanism in place to actually prevent access to the "private" attributes of a class. In particular, if the name of the class and corresponding private attribute are known, they can be accessed using the mangled name. A class can make these attributes less visible by redefining the `__dir__()` method, which supplies the list of names returned by the `dir()` function that's used to inspect objects.

Although this name mangling might look like an extra processing step, the mangling process actually only occurs once at the time a class is defined. It does not occur during execution of the methods, nor does it add extra overhead to program execution. Also, be aware that name mangling does not occur in functions such as `getattr()`, `hasattr()`, `setattr()`, or `delattr()` where the attribute name is specified as a string. For these functions, you need to explicitly use the mangled name such as `_Classname__name` to access the attribute.

It is recommended that private attributes be used when defining mutable attributes via properties. By doing so, you will encourage users to use the property name rather than accessing the underlying instance data directly (which is probably not what you intended if you wrapped it with a property to begin with). An example of this appeared in the previous section.

Giving a method a private name is a technique that a superclass can use to prevent a derived class from redefining and changing the implementation of a method. For example, the `A.bar()` method in the example only calls `A.__spam()`, regardless of the type of `self` or the presence of a different `__spam()` method in a derived class.

Finally, don't confuse the naming of private class attributes with the naming of "private" definitions in a module. A common mistake is to define a class where a single leading underscore is used on attribute names in an effort to hide their values (e.g., `_name`). In modules, this naming convention prevents names from being exported by the `from module import *` statement. However, in classes, this naming convention does not hide the attribute nor does it prevent name clashes that arise if someone inherits from the class and defines a new attribute or method with the same name.

Object Memory Management

When a class is defined, the resulting class is a factory for creating new instances. For example:

```
class Circle(object):
    def __init__(self,radius):
         self.radius = radius

# Create some Circle instances
c = Circle(4.0)
d = Circle(5.0)
```

The creation of an instance is carried out in two steps using the special method `__new__()`, which creates a new instance, and `__init__()`, which initializes it. For example, the operation `c = Circle(4.0)` performs these steps:

```
c = Circle.__new__(Circle, 4.0)
if isinstance(c,Circle):
    Circle.__init__(c,4.0)
```

The `__new__()` method of a class is something that is rarely defined by user code. If it is defined, it is typically written with the prototype `__new__(cls, *args, **kwargs)` where `args` and `kwargs` are the same arguments that will be passed to `__init__()`. `__new__()` is always a class method that receives the class object as the first parameter. Although `__new__()` creates an instance, it does not automatically call `__init__()`.

If you see `__new__()` defined in a class, it usually means the class is doing one of two things. First, the class might be inheriting from a base class whose instances are immutable. This is common if defining objects that inherit from an immutable built-in type such as an integer, string, or tuple because `__new__()` is the only method that executes prior to the instance being created and is the only place where the value could be modified (in `__init__()`, it would be too late). For example:

```
class Upperstr(str):
    def __new__(cls,value=""):
        return str.__new__(cls, value.upper())

u = Upperstr("hello")     # value is "HELLO"
```

The other major use of `__new__()` is when defining metaclasses. This is described at the end of this chapter.

Once created, instances are managed by reference counting. If the reference count reaches zero, the instance is immediately destroyed. When the instance is about to be destroyed, the interpreter first looks for a `__del__()` method associated with the object and calls it. In practice, it's rarely necessary for a class to define a `__del__()` method. The only exception is when the destruction of an object requires a cleanup action such as closing a file, shutting down a network connection, or releasing other system resources. Even in these cases, it's dangerous to rely on `__del__()` for a clean shutdown because there's no guarantee that this method will be called when the interpreter exits. A better approach may be to define a method such as `close()` that a program can use to explicitly perform a shutdown.

Occasionally, a program will use the `del` statement to delete a reference to an object. If this causes the reference count of the object to reach zero, the `__del__()` method is called. However, in general, the `del` statement doesn't directly call `__del__()`.

A subtle danger involving object destruction is that instances for which `__del__()` is defined cannot be collected by Python's cyclic garbage collector (which is a strong reason not to define `__del__` unless you need to). Programmers coming from languages without automatic garbage collection (e.g., C++) should take care not to adopt a programming style where `__del__()` is unnecessarily defined. Although it is rare to break the garbage collector by defining `__del__()`, there are certain types of programming patterns, especially those involving parent-child relationships or graphs, where this can be a problem. For example, suppose you had an object that was implementing a variant of the "Observer Pattern."

```
class Account(object):
    def __init__(self,name,balance):
         self.name = name
         self.balance = balance
         self.observers = set()
    def __del__(self):
         for ob in self.observers:
             ob.close()
         del self.observers
    def register(self,observer):
        self.observers.add(observer)
    def unregister(self,observer):
        self.observers.remove(observer)
    def notify(self):
        for ob in self.observers:
            ob.update()
    def withdraw(self,amt):
        self.balance -= amt
        self.notify()

class AccountObserver(object):
     def __init__(self, theaccount):
         self.theaccount = theaccount
         theaccount.register(self)
     def __del__(self):
         self.theaccount.unregister(self)
         del self.theaccount
     def update(self):
         print("Balance is %0.2f" % self.theaccount.balance)
     def close(self):
         print("Account no longer in use")

# Example setup
a = Account('Dave',1000.00)
a_ob = AccountObserver(a)
```

In this code, the `Account` class allows a set of `AccountObserver` objects to monitor an `Account` instance by receiving an update whenever the balance changes. To do this, each `Account` keeps a set of the observers and each `AccountObserver` keeps a reference back to the account. Each class has defined `__del__()` in an attempt to provide some sort of cleanup (such as unregistering and so on). However, it just doesn't work. Instead, the classes have created a reference cycle in which the reference count never drops to 0 and there is no cleanup. Not only that, the garbage collector (the `gc` module) won't even clean it up, resulting in a permanent memory leak.

One way to fix the problem shown in this example is for one of the classes to create a weak reference to the other using the `weakref` module. A *weak reference* is a way of creating a reference to an object without increasing its reference count. To work with a weak reference, you have to add an extra bit of functionality to check whether the object being referred to still exists. Here is an example of a modified observer class:

```
import weakref
class AccountObserver(object):
     def __init__(self, theaccount):
         self.accountref = weakref.ref(theaccount)  # Create a weakref
         theaccount.register(self)
```

```
    def __del__(self):
        acc = self.accountref()        # Get account
        if acc:                        # Unregister if still exists
            acc.unregister(self)
    def update(self):
        print("Balance is %0.2f" % self.accountref().balance)
    def close(self):
        print("Account no longer in use")

# Example setup
a = Account('Dave',1000.00)
a_ob = AccountObserver(a)
```

In this example, a weak reference `accountref` is created. To access the underlying `Account`, you call it like a function. This either returns the `Account` or `None` if it's no longer around. With this modification, there is no longer a reference cycle. If the `Account` object is destroyed, its `__del__` method runs and observers receive notification. The `gc` module also works properly. More information about the `weakref` module can be found in Chapter 13, "Python Runtime Services."

Object Representation and Attribute Binding

Internally, instances are implemented using a dictionary that's accessible as the instance's `__dict__` attribute. This dictionary contains the data that's unique to each instance. Here's an example:

```
>>> a = Account('Guido', 1100.0)
>>> a.__dict__
{'balance': 1100.0, 'name': 'Guido'}
```

New attributes can be added to an instance at any time, like this:

```
a.number = 123456    # Add attribute 'number' to a.__dict__
```

Modifications to an instance are always reflected in the local `__dict__` attribute. Likewise, if you make modifications to `__dict__` directly, those modifications are reflected in the attributes.

Instances are linked back to their class by a special attribute `__class__`. The class itself is also just a thin layer over a dictionary which can be found in its own `__dict__` attribute. The class dictionary is where you find the methods. For example:

```
>>> a.__class__
<class '__main__.Account'>
>>> Account.__dict__.keys()
['__dict__', '__module__', 'inquiry', 'deposit', 'withdraw',
'__del__', 'num_accounts', '__weakref__', '__doc__', '__init__']
>>>
```

Finally, classes are linked to their base classes in a special attribute `__bases__`, which is a tuple of the base classes. This underlying structure is the basis for all of the operations that get, set, and delete the attributes of objects.

Whenever an attribute is set using `obj.`*`name`*` = `*`value`*, the special method `obj.__setattr__("`*`name`*`", `*`value`*`)` is invoked. If an attribute is deleted using `del obj.`*`name`*, the special method `obj.__delattr__("`*`name`*`")` is invoked. The default behavior of these methods is to modify or remove values from the local `__dict__` of `obj` unless the requested attribute happens to correspond to a property or descriptor. In that case, the set and delete operation will be carried out by the set and delete functions associated with the property.

For attribute lookup such as `obj.name`, the special method `obj.__getattrribute__("name")` is invoked. This method carries out the search process for finding the attribute, which normally includes checking for properties, looking in the local `__dict__` attribute, checking the class dictionary, and searching the base classes. If this search process fails, a final attempt to find the attribute is made by trying to invoke the `__getattr__()` method of the class (if defined). If this fails, an `AttributeError` exception is raised.

User-defined classes can implement their own versions of the attribute access functions, if desired. For example:

```
class Circle(object):
    def __init__(self,radius):
        self.radius = radius
    def __getattr__(self,name):
        if name == 'area':
             return math.pi*self.radius**2
        elif name == 'perimeter':
             return 2*math.pi*self.radius
        else:
             return object.__getattr__(self,name)
    def __setattr__(self,name,value):
        if name in ['area','perimeter']:
             raise TypeError("%s is readonly" % name)
        object.__setattr__(self,name,value)
```

A class that reimplements these methods should probably rely upon the default implementation in `object` to carry out the actual work. This is because the default implementation takes care of the more advanced features of classes such as descriptors and properties.

As a general rule, it is relatively uncommon for classes to redefine the attribute access operators. However, one application where they are often used is in writing general-purpose wrappers and proxies to existing objects. By redefining `__getattr__()`, `__setattr__()`, and `__delattr__()`, a proxy can capture attribute access and transparently forward those operations on to another object.

`__slots__`

A class can restrict the set of legal instance attribute names by defining a special variable called `__slots__`. Here's an example:

```
class Account(object):
    __slots__ = ('name','balance')
      ...
```

When `__slots__` is defined, the attribute names that can be assigned on instances are restricted to the names specified. Otherwise, an `AttributeError` exception is raised. This restriction prevents someone from adding new attributes to existing instances and solves the problem that arises if someone assigns a value to an attribute that they can't spell correctly.

In reality, `__slots__` was never implemented to be a safety feature. Instead, it is actually a performance optimization for both memory and execution speed. Instances of a class that uses `__slots__` no longer use a dictionary for storing instance data. Instead, a much more compact data structure based on an array is used. In programs that create a large number of objects, using `__slots__` can result in a substantial reduction in memory use and execution time.

Be aware that the use of `__slots__` has a tricky interaction with inheritance. If a class inherits from a base class that uses `__slots__`, it also needs to define `__slots__` for storing its own attributes (even if it doesn't add any) to take advantage of the benefits `__slots__` provides. If you forget this, the derived class will run slower and use even more memory than what would have been used if `__slots__` had not been used on *any* of the classes!

The use of `__slots__` can also break code that expects instances to have an underlying `__dict__` attribute. Although this often does not apply to user code, utility libraries and other tools for supporting objects may be programmed to look at `__dict__` for debugging, serializing objects, and other operations.

Finally, the presence of `__slots__` has no effect on the invocation of methods such as `__getattribute__()`, `__getattr__()`, and `__setattr__()` should they be redefined in a class. However, the default behavior of these methods will take `__slots__` into account. In addition, it should be stressed that it is not necessary to add method or property names to `__slots__`, as they are stored in the class, not on a per-instance basis.

Operator Overloading

User-defined objects can be made to work with all of Python's built-in operators by adding implementations of the special methods described in Chapter 3 to a class. For example, if you wanted to add a new kind of number to Python, you could define a class in which special methods such as `__add__()` were defined to make instances work with the standard mathematical operators.

The following example shows how this works by defining a class that implements the complex numbers with some of the standard mathematical operators.

Note

Because Python already provides a complex number type, this class is only provided for the purpose of illustration.

```
class Complex(object):
    def __init__(self,real,imag=0):
        self.real = float(real)
        self.imag = float(imag)
    def __repr__(self):
        return "Complex(%s,%s)" % (self.real, self.imag)
    def __str__(self):
        return "(%g+%gj)" % (self.real, self.imag)
    # self + other
    def __add__(self,other):
        return Complex(self.real + other.real, self.imag + other.imag)
    # self - other
    def __sub__(self,other):
        return Complex(self.real - other.real, self.imag - other.imag)
```

In the example, the `__repr__()` method creates a string that can be evaluated to re-create the object (that is, `"Complex(`*`real`*`,`*`imag`*`)"`). This convention should be followed for all user-defined objects as applicable. On the other hand, the `__str__()` method creates a string that's intended for nice output formatting (this is the string that would be produced by the `print` statement).

The other operators, such as `__add__()` and `__sub__()`, implement mathematical operations. A delicate matter with these operators concerns the order of operands and type coercion. As implemented in the previous example, the `__add__()` and `__sub__()` operators are applied *only* if a complex number appears on the left side of the operator. They do not work if they appear on the right side of the operator and the left-most operand is not a `Complex`. For example:

```
>>> c = Complex(2,3)
>>> c + 4.0
Complex(6.0,3.0)
>>> 4.0 + c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'Complex'

>>>
```

The operation `c + 4.0` works partly by accident. All of Python's built-in numbers already have `.real` and `.imag` attributes, so they were used in the calculation. If the `other` object did not have these attributes, the implementation would break. If you want your implementation of `Complex` to work with objects missing these attributes, you have to add extra conversion code to extract the needed information (which might depend on the type of the other object).

The operation `4.0 + c` does not work at all because the built-in floating point type doesn't know anything about the `Complex` class. To fix this, you can add reversed-operand methods to `Complex`:

```
class Complex(object):
    ...
    def __radd__(self,other):
        return Complex(other.real + self.real, other.imag + self.imag)
    def __rsub__(self,other):
        return Complex(other.real - self.real, other.imag - self.img)
    ...
```

These methods serve as a fallback. If the operation `4.0 + c` fails, Python tries to execute `c.__radd__(4.0)` first before issuing a `TypeError`.

Older versions of Python have tried various approaches to coerce types in mixed-type operations. For example, you might encounter legacy Python classes that implement a `__coerce__()` method. This is no longer used by Python 2.6 or Python 3. Also, don't be fooled by special methods such as `__int__()`, `__float__()`, or `__complex__()`. Although these methods are called by explicit conversions such as `int(x)` or `float(x)`, they are never called implicitly to perform type conversion in mixed-type arithmetic. So, if you are writing classes where operators must work with mixed types, you have to explicitly handle the type conversion in the implementation of each operator.

Types and Class Membership Tests

When you create an instance of a class, the type of that instance is the class itself. To test for membership in a class, use the built-in function `isinstance(obj,cname)`. This function returns `True` if an object, *`obj`*, belongs to the class *`cname`* or any class derived from *`cname`*. Here's an example:

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

a = A()          # Instance of 'A'
b = B()          # Instance of 'B'
c = C()          # Instance of 'C'

type(a)          # Returns the class object A
isinstance(a,A)  # Returns True
isinstance(b,A)  # Returns True, B derives from A
isinstance(b,C)  # Returns False, C not derived from A
```

Similarly, the built-in function `issubclass(`*`A`*`,`*`B`*`)` returns `True` if the class *`A`* is a subclass of class *`B`*. Here's an example:

```
issubclass(B,A)   # Returns True
issubclass(C,A)   # Returns False
```

A subtle problem with type-checking of objects is that programmers often bypass inheritance and simply create objects that mimic the behavior of another object. As an example, consider these two classes:

```
class Foo(object):
    def spam(self,a,b):
        pass

class FooProxy(object):
    def __init__(self,f):
        self.f = f
    def spam(self,a,b):
        return self.f.spam(a,b)
```

In this example, `FooProxy` is functionally identical to `Foo`. It implements the same methods, and it even uses `Foo` underneath the covers. Yet, in the type system, `FooProxy` is different than `Foo`. For example:

```
f = Foo()           # Create a Foo
g = FooProxy(f)     # Create a FooProxy
isinstance(g, Foo)  # Returns False
```

If a program has been written to explicitly check for a `Foo` using `isinstance()`, then it certainly won't work with a `FooProxy` object. However, this degree of strictness is often not exactly what you want. Instead, it might make more sense to assert that an object can simply be used as `Foo` because it has the same interface. To do this, it is possible to define an object that redefines the behavior of `isinstance()` and `issubclass()` for the purpose of grouping objects together and type-checking. Here is an example:

```
class IClass(object):
    def __init__(self):
        self.implementors = set()
    def register(self,C):
        self.implementors.add(C)
    def __instancecheck__(self,x):
        return self.__subclasscheck__(type(x))
    def __subclasscheck__(self,sub):
         return any(c in self.implementors for c in sub.mro())

# Now, use the above object
IFoo = IClass()
IFoo.register(Foo)
IFoo.register(FooProxy)
```

In this example, the class `IClass` creates an object that merely groups a collection of other classes together in a set. The `register()` method adds a new class to the set. The special method `__instancecheck__()` is called if anyone performs the operation `isinstance(x, IClass)`. The special method `__subclasscheck__()` is called if the operation `issubclass(C,IClass)` is called.

By using the `IFoo` object and registered implementers, one can now perform type checks such as the following:

```
f = Foo()           # Create a Foo
g = FooProxy(f)     # Create a FooProxy
isinstance(f, IFoo)        # Returns True
isinstance(g, IFoo)        # Returns True
issubclass(FooProxy, IFoo) # Returns True
```

In this example, it's important to emphasize that no strong type-checking is occurring. The `IFoo` object has overloaded the instance checking operations in a way that allows a you to assert that a class belongs to a group. It doesn't assert any information on the actual programming interface, and no other verification actually occurs. In fact, you can simply register any collection of objects you want to group together without regard to how those classes are related to each other. Typically, the grouping of classes is based on some criteria such as all classes implementing the same programming interface. However, no such meaning should be inferred when overloading `__instancecheck__()` or `__subclasscheck__()`. The actual interpretation is left up to the application.

Python provides a more formal mechanism for grouping objects, defining interfaces, and type-checking. This is done by defining an abstract base class, which is defined in the next section.

Abstract Base Classes

In the last section, it was shown that the `isinstance()` and `issubclass()` operations can be overloaded. This can be used to create objects that group similar classes together and to perform various forms of type-checking. *Abstract base classes* build upon this concept and provide a means for organizing objects into a hierarchy, making assertions about required methods, and so forth.

To define an abstract base class, you use the `abc` module. This module defines a metaclass (`ABCMeta`) and a set of decorators (`@abstractmethod` and `@abstractproperty`) that are used as follows:

```
from abc import ABCMeta, abstractmethod, abstractproperty
class Foo:                    # In Python 3, you use the syntax
    __metaclass__ = ABCMeta   # class Foo(metaclass=ABCMeta)
    @abstractmethod
    def spam(self,a,b):
        pass
    @abstractproperty
    def name(self):
        pass
```

The definition of an abstract class needs to set its metaclass to `ABCMeta` as shown (also, be aware that the syntax differs between Python 2 and 3). This is required because the implementation of abstract classes relies on a metaclass (described in the next section). Within the abstract class, the `@abstractmethod` and `@abstractproperty` decorators specify that a method or property must be implemented by subclasses of `Foo`.

An abstract class is not meant to be instantiated directly. If you try to create a `Foo` for the previous class, you will get the following error:

```
>>> f = Foo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Foo with abstract methods spam
>>>
```

This restriction carries over to derived classes as well. For instance, if you have a class `Bar` that inherits from `Foo` but it doesn't implement one or more of the abstract methods, attempts to create a `Bar` will fail with a similar error. Because of this added checking, abstract classes are useful to programmers who want to make assertions on the methods and properties that must be implemented on subclasses.

Although an abstract class enforces rules about methods and properties that must be implemented, it does not perform conformance checking on arguments or return values. Thus, an abstract class will not check a subclass to see whether a method has used the same arguments as an abstract method. Likewise, an abstract class that requires the definition of a property does not check to see whether the property in a subclass supports the same set of operations (`get`, `set`, and `delete`) of the property specified in a base.

Although an abstract class can not be instantiated, it can define methods and properties for use in subclasses. Moreover, an abstract method in the base can still be called from a subclass. For example, calling `Foo.spam(a,b)` from the subclass is allowed.

Abstract base classes allow preexisting classes to be registered as belonging to that base. This is done using the `register()` method as follows:

```
class Grok(object):
    def spam(self,a,b):
        print("Grok.spam")

Foo.register(Grok)       # Register with Foo abstract base class
```

When a class is registered with an abstract base, type-checking operations involving the abstract base (such as `isinstance()` and `issubclass()`) will return `True` for instances of the registered class. When a class is registered with an abstract class, no checks are made to see whether the class actually implements any of the abstract methods or properties. This registration process only affects type-checking. It does not add extra error checking to the class that is registered.

Unlike many other object-oriented languages, Python's built-in types are organized into a relatively flat hierarchy. For example, if you look at the built-in types such as `int` or `float`, they directly inherit from `object`, the root of all objects, instead of an intermediate base class representing numbers. This makes it clumsy to write programs that want to inspect and manipulate objects based on a generic category such as simply being an instance of a number.

The abstract class mechanism addresses this issue by allowing preexisting objects to be organized into user-definable type hierarchies. Moreover, some library modules aim to organize the built-in types according to different capabilities that they possess. The `collections` module contains abstract base classes for various kinds of operations involving sequences, sets, and dictionaries. The `numbers` module contains abstract base classes related to organizing a hierarchy of numbers. Further details can be found in Chapter 14, "Mathematics," and Chapter 15, "Data Structures, Algorithms, and Utilities."

Metaclasses

When you define a class in Python, the class definition itself becomes an object. Here's an example:

```
class Foo(object): pass
isinstance(Foo,object)        # Returns True
```

If you think about this long enough, you will realize that something had to create the `Foo` object. This creation of the class object is controlled by a special kind of object called a *metaclass*. Simply stated, a metaclass is an object that knows how to create and manage classes.

In the preceding example, the metaclass that is controlling the creation of `Foo` is a class called `type`. In fact, if you display the type of `Foo`, you will find out that it *is* a `type`:

```
>>> type(Foo)
<type 'type'>
```

When a new class is defined with the `class` statement, a number of things happen. First, the body of the class is executed as a series of statements within its own private dictionary. The execution of statements is exactly the same as in normal code with the addition of the name mangling that occurs on private members (names that start with __). Finally, the name of the class, the list of base classes, and the dictionary are passed to the constructor of a metaclass to create the corresponding class object. Here is an example of how it works:

```
class_name = "Foo"                 # Name of class
class_parents = (object,)          # Base classes
class_body = """                   # Class body
def __init__(self,x):
    self.x = x
def blah(self):
    print("Hello World")
"""
class_dict = { }
# Execute the body in the local dictionary class_dict
exec(class_body,globals(),class_dict)

# Create the class object Foo
Foo = type(class_name,class_parents,class_dict)
```

The final step of class creation where the metaclass `type()` is invoked can be customized. The choice of what happens in the final step of class definition is controlled in a number of ways. First, the class can explicitly specify its metaclass by either setting a `__metaclass__` class variable (Python 2), or supplying the `metaclass` keyword argument in the tuple of base classes (Python 3).

```
class Foo:                  # In Python 3, use the syntax
    __metaclass__ = type    # class Foo(metaclass=type)
    ...
```

If no metaclass is explicitly specified, the `class` statement examines the first entry in the tuple of base classes (if any). In this case, the metaclass is the same as the type of the first base class. Therefore, when you write

```
class Foo(object): pass
```

`Foo` will be the same type of class as `object`.

If no base classes are specified, the `class` statement checks for the existence of a global variable called `__metaclass__`. If this variable is found, it will be used to create classes. If you set this variable, it will control how classes are created when a simple class statement is used. Here's an example:

```
__metaclass__ = type
class Foo:
      pass
```

Finally, if no `__metaclass__` value can be found anywhere, Python uses the default metaclass. In Python 2, this defaults to `types.ClassType`, which is known as an *old-style class*. This kind of class, deprecated since Python 2.2, corresponds to the original implementation of classes in Python. Although these classes are still supported, they should be avoided in new code and are not covered further here. In Python 3, the default metaclass is simply `type()`.

The primary use of metaclasses is in frameworks that want to assert more control over the definition of user-defined objects. When a custom metaclass is defined, it typically inherits from `type()` and reimplements methods such as `__init__()` or `__new__()`. Here is an example of a metaclass that forces all methods to have a documentation string:

```
class DocMeta(type):
    def __init__(self,name,bases,dict):
        for key, value in dict.items():
            # Skip special and private methods
            if key.startswith("__"): continue
            # Skip anything not callable
            if not hasattr(value,"__call__"): continue
            # Check for a doc-string
            if not getattr(value,"__doc__"):
                raise TypeError("%s must have a docstring" % key)
        type.__init__(self,name,bases,dict)
```

In this metaclass, the `__init__()` method has been written to inspect the contents of the class dictionary. It scans the dictionary looking for methods and checking to see whether they all have documentation strings. If not, a `TypeError` exception is generated. Otherwise, the default implementation of `type.__init__()` is called to initialize the class.

To use this metaclass, a class needs to explicitly select it. The most common technique for doing this is to first define a base class such as the following:

```
class Documented:               # In Python 3, use the syntax
    __metaclass__ = DocMeta     # class Documented(metaclass=DocMeta)
```

This base class is then used as the parent for all objects that are to be documented. For example:

```
class Foo(Documented):
    spam(self,a,b):
        "spam does something"
        pass
```

This example illustrates one of the major uses of metaclasses, which is that of inspecting and gathering information about class definitions. The metaclass isn't changing anything about the class that actually gets created but is merely adding some additional checks.

In more advanced metaclass applications, a metaclass can both inspect and alter the contents of a class definition prior to the creation of the class. If alterations are going to be made, you should redefine the `__new__()` method that runs prior to the creation of the class itself. This technique is commonly combined with techniques that wrap attributes with descriptors or properties because it is one way to capture the names being used in the class. As an example, here is a modified version of the `TypedProperty` descriptor that was used in the "Descriptors" section:

```
class TypedProperty(object):
    def __init__(self,type,default=None):
        self.name = None
        self.type = type
        if default: self.default = default
        else:       self.default = type()
    def __get__(self,instance,cls):
        return getattr(instance,self.name,self.default)
    def __set__(self,instance,value):
        if not isinstance(value,self.type):
            raise TypeError("Must be a %s" % self.type)
        setattr(instance,self.name,value)
    def __delete__(self,instance):
        raise AttributeError("Can't delete attribute")
```

In this example, the `name` attribute of the descriptor is simply set to `None`. To fill this in, we'll rely on a meta class. For example:

```
class TypedMeta(type):
    def __new__(cls,name,bases,dict):
        slots = []
        for key,value in dict.items():
            if isinstance(value,TypedProperty):
                value.name = "_" + key
                slots.append(value.name)
        dict['__slots__'] = slots
        return type.__new__(cls,name,bases,dict)

# Base class for user-defined objects to use
class Typed:                      # In Python 3, use the syntax
    __metaclass__ = TypedMeta    # class Typed(metaclass=TypedMeta)
```

In this example, the metaclass scans the class dictionary and looks for instances of `TypedProperty`. If found, it sets the `name` attribute and builds a list of names in `slots`. After this is done, a `__slots__` attribute is added to the class dictionary, and the class is constructed by calling the `__new__()` method of the `type()` metaclass. Here is an example of using this new metaclass:

```
class Foo(Typed):
    name = TypedProperty(str)
    num  = TypedProperty(int,42)
```

Although metaclasses make it possible to drastically alter the behavior and semantics of user-defined classes, you should probably resist the urge to use metaclasses in a way that makes classes work wildly different from what is described in the standard Python documentation. Users will be confused if the classes they must write don't adhere to any of the normal coding rules expected for classes.

Class Decorators

In the previous section, it was shown how the process of creating a class can be customized by defining a metaclass. However, sometimes all you want to do is perform some kind of extra processing after a class is defined, such as adding a class to a registry or database. An alternative approach for such problems is to use a class decorator. A *class decorator* is a function that takes a class as input and returns a class as output. For example:

```
registry = { }
def register(cls):
    registry[cls.__clsid__] = cls
    return cls
```

In this example, the register function looks inside a class for a `__clsid__` attribute. If found, it's used to add the class to a dictionary mapping class identifiers to class objects. To use this function, you can use it as a decorator right before the class definition. For example:

```
@register
class Foo(object):
    __clsid__ = "123-456"
    def bar(self):
        pass
```

Here, the use of the decorator syntax is mainly one of convenience. An alternative way to accomplish the same thing would have been this:

```
class Foo(object):
    __clsid__ = "123-456"
    def bar(self):
        pass
register(Foo)       # Register the class
```

Although it's possible to think of endless diabolical things one might do to a class in a class decorator function, it's probably best to avoid excessive magic such as putting a wrapper around the class or rewriting the class contents.

This page intentionally left blank