CHAPTER 6

Oh Oh: Objects and Classes

No object is mysterious. The mystery is your eye.
—Elizabeth Bowen

Take an object. Do something to it. Do something else to it.
—Jasper Johns

Up to this point, you've seen data structures such as strings and dictionaries, and code structures such as functions and modules. In this chapter, you'll deal with custom data structures: *objects*.

What Are Objects?

As I mention in Chapter 2, everything in Python, from numbers to modules, is an object. However, Python hides most of the object machinery by means of special syntax. You can type `num = 7` to create a object of type integer with the value 7, and assign an object reference to the name `num`. The only time you need to look inside objects is when you want to make your own or modify the behavior of existing objects. You'll see how to do both in this chapter.

An object contains both data (variables, called *attributes*) and code (functions, called *methods*). It represents a unique instance of some concrete thing. For example, the integer object with the value `7` is an object that facilitates methods such as addition and multiplication, as is demonstrated in "Numbers" on page 21. `8` is a different object. This means there's an Integer class in Python, to which both `7` and `8` belong. The strings `'cat'` and `'duck'` are also objects in Python, and have string methods that you've seen, such as `capitalize()` and `replace()`.

When you create new objects no one has ever created before, you must create a class that indicates what they contain.

Think of objects as nouns and their methods as verbs. An object represents an individual thing, and its methods define how it interacts with other things.

Unlike modules, you can have multiple objects at the same time, each one with different values for its attributes. They're like super data structures, with code thrown in.

Define a Class with class

In Chapter 1, I compare an object to a plastic box. A *class* is like the mold that makes that box. For instance, a `String` is the built-in Python class that makes string objects such as `'cat'` and `'duck'`. Python has many other built-in classes to create the other standard data types, including lists, dictionaries, and so on. To create your own custom object in Python, you first need to define a class by using the `class` keyword. Let's walk through a simple example.

Suppose that you want to define objects to represent information about people. Each object will represent one person. You'll first want to define a class called `Person` as the mold. In the examples that follow, we'll try more than one version of this class as we build up from the simplest class to ones that actually do something useful.

Our first try is the simplest possible class, an empty one:

```
>>> class Person():
...     pass
```

Just as with functions, we needed to say `pass` to indicate that this class was empty. This definition is the bare minimum to create an object. You create an object from a class by calling the class name as though it were a function:

```
>>> someone = Person()
```

In this case, `Person()` creates an individual object from the `Person` class and assigns it the name `someone`. But, our `Person` class was empty, so the `someone` object that we create from it just sits there and can't do anything else. You would never actually define such a class, and I'm only showing it here to build up to the next example.

Let's try again, this time including the special Python object initialization method `__init__`:

```
>>> class Person():
...     def __init__(self):
...         pass
```

This is what you'll see in real Python class definitions. I admit that the `__init__()` and `self` look strange. `__init__()` is the special Python name for a method that initializes an individual object from its class definition. [1] The `self` argument specifies that it refers to the individual object itself.

When you define `__init__()` in a class definition, its first parameter should be `self`. Although `self` is not a reserved word in Python, it's common usage. No one reading your code later (including you!) will need to guess what you meant if you use `self`.

But even that second `Person` definition didn't create an object that really did anything. The third try is the charm that really shows how to create a simple object in Python. This time, we'll add the parameter `name` to the initialization method:

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...
>>>
```

Now, we can create an object from the `Person` class by passing a string for the `name` parameter:

```
>>> hunter = Person('Elmer Fudd')
```

Here's what this line of code does:

- Looks up the definition of the `Person` class
- *Instantiates* (creates) a new object in memory
- Calls the object's `__init__` method, passing this newly-created object as `self` and the other argument (`'Elmer Fudd'`) as `name`
- Stores the value of `name` in the object
- Returns the new object
- Attaches the name `hunter` to the object

This new object is like any other object in Python. You can use it as an element of a list, tuple, dictionary, or set. You can pass it to a function as an argument, or return it as a result.

What about the `name` value that we passed in? It was saved with the object as an attribute. You can read and write it directly:

```
>>> print('The mighty hunter: ', hunter.name)
The mighty hunter: Elmer Fudd
```

1 You'll see many examples of double underscores in Python names; to save syllables, some people pronounce them as *dunder*.

Remember, *inside* the `Person` class definition, you access the `name` attribute as `self.name`. When you create an actual object such as `hunter`, you refer to it as `hunter.name`.

It is *not* necessary to have an `__init__` method in every class definition; it's used to do anything that's needed to distinguish this object from others created from the same class.

Inheritance

When you're trying to solve some coding problem, often you'll find an existing class that creates objects that do almost what you need. What can you do? You could modify this old class, but you'll make it more complicated, and you might break something that used to work.

Of course, you could write a new class, cutting and pasting from the old one and merging your new code. But this means that you have more code to maintain, and the parts of the old and new classes that used to work the same might drift apart because they're now in separate places.

The solution is *inheritance*: creating a new class from an existing class but with some additions or changes. It's an excellent way to reuse code. When you use inheritance, the new class can automatically use all the code from the old class but without copying any of it.

You define only what you need to add or change in the new class, and this overrides the behavior of the old class. The original class is called a *parent*, *superclass*, or *base class*; the new class is called a *child*, *subclass*, or *derived class*. These terms are interchangeable in object-oriented programming.

So, let's inherit something. We'll define an empty class called `Car`. Next, define a subclass of `Car` called `Yugo`. You define a subclass by using the same `class` keyword but with the parent class name inside the parentheses (`class Yugo(Car)` below):

```
>>> class Car():
...     pass
...
>>> class Yugo(Car):
...     pass
...
```

Next, create an object from each class:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

A child class is a specialization of a parent class; in object-oriented lingo, `Yugo` *is-a* `Car`. The object named `give_me_a_yugo` is an instance of class `Yugo`, but it also inherits whatever a `Car` can do. In this case, `Car` and `Yugo` are as useful as deckhands on a submarine, so let's try new class definitions that actually do something:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     pass
...
```

Finally, make one object from each class and call the `exclaim` method:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
>>> give_me_a_car.exclaim()
I'm a Car!
>>> give_me_a_yugo.exclaim()
I'm a Car!
```

Without doing anything special, `Yugo` inherited the `exclaim()` method from `Car`. In fact, `Yugo` says that it *is* a `Car`, which might lead to an identity crisis. Let's see what we can do about that.

Override a Method

As you just saw, a new class initially inherits everything from its parent class. Moving forward, you'll see how to replace or *override* a parent method. `Yugo` should probably be different from `Car` in some way; otherwise, what's the point of defining a new class? Let's change how the `exclaim()` method works for a `Yugo`:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     def exclaim(self):
...         print("I'm a Yugo! Much like a Car, but more Yugo-ish.")
...
```

Now, make two objects from these classes:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

What do they say?

```
>>> give_me_a_car.exclaim()
I'm a Car!
>>> give_me_a_yugo.exclaim()
I'm a Yugo! Much like a Car, but more Yugo-ish.
```

In these examples, we overrode the `exclaim()` method. We can override any methods, including `__init__()`. Here's another example that uses our earlier `Person` class. Let's make subclasses that represent doctors (`MDPerson`) and lawyers (`JDPerson`):

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...
>>> class MDPerson(Person):
...     def __init__(self, name):
...         self.name = "Doctor " + name
...
>>> class JDPerson(Person):
...     def __init__(self, name):
...         self.name = name + ", Esquire"
...
```

In these cases, the initialization method `__init__()` takes the same arguments as the parent `Person` class but stores the value of `name` differently inside the object instance:

```
>>> person = Person('Fudd')
>>> doctor = MDPerson('Fudd')
>>> lawyer = JDPerson('Fudd')
>>> print(person.name)
Fudd
>>> print(doctor.name)
Doctor Fudd
>>> print(lawyer.name)
Fudd, Esquire
```

Add a Method

The child class can also *add* a method that was not present in its parent class. Going back to classes `Car` and `Yugo`, we'll define the new method `need_a_push()` for class `Yugo` only:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     def exclaim(self):
...         print("I'm a Yugo! Much like a Car, but more Yugo-ish.")
...     def need_a_push(self):
...         print("A little help here?")
...
```

Next, make a `Car` and a `Yugo`:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

A Yugo object can react to a need_a_push() method call:

```
>>> give_me_a_yugo.need_a_push()
A little help here?
```

But a generic Car object cannot:

```
>>> give_me_a_car.need_a_push()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Car' object has no attribute 'need_a_push'
```

At this point, a Yugo can do something that a Car cannot, and the distinct personality of a Yugo can emerge.

Get Help from Your Parent with super

We saw how the child class could add or override a method from the parent. What if it wanted to call that parent method? "I'm glad you asked," says super(). We'll define a new class called EmailPerson that represents a Person with an email address. First, our familiar Person definition:

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...
```

Notice that the __init__() call in the following subclass has an additional email parameter:

```
>>> class EmailPerson(Person):
...     def __init__(self, name, email):
...         super().__init__(name)
...         self.email = email
```

When you define an __init__() method for your class, you're replacing the __init__() method of its parent class, and the latter is not called automatically anymore. As a result, we need to call it explicitly. Here's what's happening:

- The super() gets the definition of the parent class, Person.
- The __init__() method calls the Person.__init__() method. It takes care of passing the self argument to the superclass, so you just need to give it any optional arguments. In our case, the only other argument Person() accepts is name.
- The self.email = email line is the new code that makes this EmailPerson different from a Person.

Moving on, let's make one of these creatures:

```
>>> bob = EmailPerson('Bob Frapples', 'bob@frapples.com')
```

We should be able to access both the `name` and `email` attributes:

```
>>> bob.name
'Bob Frapples'
>>> bob.email
'bob@frapples.com'
```

Why didn't we just define our new class as follows?

```
>>> class EmailPerson(Person):
...     def __init__(self, name, email):
...         self.name = name
...         self.email = email
```

We could have done that, but it would have defeated our use of inheritance. We used `super()` to make `Person` do its work, the same as a plain `Person` object would. There's another benefit: if the definition of `Person` changes in the future, using `super()` will ensure that the attributes and methods that `EmailPerson` inherits from `Person` will reflect the change.

Use `super()` when the child is doing something its own way but still needs something from the parent (as in real life).

In self Defense

One criticism of Python (besides the use of whitespace) is the need to include `self` as the first argument to instance methods (the kind of method you've seen in the previous examples). Python uses the `self` argument to find the right object's attributes and methods. For an example, I'll show how you would call an object's method, and what Python actually does behind the scenes.

Remember class `Car` from earlier examples? Let's call its `exclaim()` method again:

```
>>> car = Car()
>>> car.exclaim()
I'm a Car!
```

Here's what Python actually does, under the hood:

- Look up the class (`Car`) of the object `car`.
- Pass the object `car` to the `exclaim()` method of the `Car` class as the `self` parameter.

Just for fun, you can even run it this way yourself and it will work the same as the normal (`car.exclaim()`) syntax:

```
>>> Car.exclaim(car)
I'm a Car!
```

However, there's never a reason to use that lengthier style.

Get and Set Attribute Values with Properties

Some object-oriented languages support private object attributes that can't be accessed directly from the outside; programmers often need to write *getter* and *setter* methods to read and write the values of such private attributes.

Python doesn't need getters or setters, because all attributes and methods are public, and you're expected to behave yourself. If direct access to attributes makes you nervous, you can certainly write getters and setters. But be Pythonic—use *properties*.

In this example, we'll define a `Duck` class with a single attribute called `hidden_name`. (In the next section, I'll show you a better way to name attributes that you want to keep private.) We don't want people to access this directly, so we'll define two methods: a getter (`get_name()`) and a setter (`set_name()`). I've added a `print()` statement to each method to show when it's being called. Finally, we define these methods as properties of the `name` attribute:

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.hidden_name = input_name
...     def get_name(self):
...         print('inside the getter')
...         return self.hidden_name
...     def set_name(self, input_name):
...         print('inside the setter')
...         self.hidden_name = input_name
...     name = property(get_name, set_name)
```

The new methods act as normal getters and setters until that last line; it defines the two methods as properties of the attribute called `name`. The first argument to `property()` is the getter method, and the second is the setter. Now, when you refer to the `name` of any `Duck` object, it actually calls the `get_name()` method to return it:

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
```

You can still call `get_name()` directly, too, like a normal getter method:

```
>>> fowl.get_name()
inside the getter
'Howard'
```

When you assign a value to the `name` attribute, the `set_name()` method will be called:

```
>>> fowl.name = 'Daffy'
inside the setter
```

```
>>> fowl.name
inside the getter
'Daffy'
```

You can still call the `set_name()` method directly:

```
>>> fowl.set_name('Daffy')
inside the setter
>>> fowl.name
inside the getter
'Daffy'
```

Another way to define properties is with *decorators*. In this next example, we'll define two different methods, each called `name()` but preceded by different decorators:

- `@property`, which goes before the getter method
- `@name.setter`, which goes before the setter method

Here's how they actually look in the code:

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.hidden_name = input_name
...     @property
...     def name(self):
...         print('inside the getter')
...         return self.hidden_name
...     @name.setter
...     def name(self, input_name):
...         print('inside the setter')
...         self.hidden_name = input_name
```

You can still access `name` as though it were an attribute, but there are no visible `get_name()` or `set_name()` methods:

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
>>> fowl.name = 'Donald'
inside the setter
>>> fowl.name
inside the getter
'Donald'
```

> If anyone guessed that we called our attribute `hidden_name`, they could still read and write it directly as `fowl.hidden_name`. In the next section, you'll see how Python provides a special way to name private attributes.

In both of the previous examples, we used the `name` property to refer to a single attribute (ours was called `hidden_name`) stored within the object. A property can refer to a *computed value*, as well. Let's define a `Circle` class that has a `radius` attribute and a computed `diameter` property:

```
>>> class Circle():
...     def __init__(self, radius):
...         self.radius = radius
...     @property
...     def diameter(self):
...         return 2 * self.radius
...
```

We create a `Circle` object with an initial value for its `radius`:

```
>>> c = Circle(5)
>>> c.radius
5
```

We can refer to `diameter` as if it were an attribute such as `radius`:

```
>>> c.diameter
10
```

Here's the fun part: we can change the `radius` attribute at any time, and the diameter property will be computed from the current value of `radius`:

```
>>> c.radius = 7
>>> c.diameter
14
```

If you don't specify a setter property for an attribute, you can't set it from the outside. This is handy for read-only attributes:

```
>>> c.diameter = 20
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

There's one more big advantage of using a property over direct attribute access: if you ever change the definition of the attribute, you only need to fix the code within the class definition, not in all the callers.

Name Mangling for Privacy

In the `Duck` class example in the previous section, we called our (not completely) hidden attribute `hidden_name`. Python has a naming convention for attributes that should not be visible outside of their class definition: begin by using with two underscores (`__`).

Let's rename `hidden_name` to `__name`, as demonstrated here:

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.__name = input_name
...     @property
...     def name(self):
...         print('inside the getter')
...         return self.__name
...     @name.setter
...     def name(self, input_name):
...         print('inside the setter')
...         self.__name = input_name
...
```

Take a moment to see if everything still works:

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
>>> fowl.name = 'Donald'
inside the setter
>>> fowl.name
inside the getter
'Donald'
```

Looks good. And, you can't access the `__name` attribute:

```
>>> fowl.__name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Duck' object has no attribute '__name'
```

This naming convention doesn't make it private, but Python does *mangle* the name to make it unlikely for external code to stumble upon it. If you're curious and promise not to tell everyone, here's what it becomes:

```
>>> fowl._Duck__name
'Donald'
```

Notice that it didn't print `inside the getter`. Although this isn't perfect protection, name mangling discourages accidental or intentional direct access to the attribute.

Method Types

Some data (*attributes*) and functions (*methods*) are part of the class itself, and some are part of the objects that are created from that class.

When you see an initial `self` argument in methods within a class definition, it's an *instance method*. These are the types of methods that you would normally write when creating your own classes. The first parameter of an instance method is `self`, and Python passes the object to the method when you call it.

In contrast, a *class method* affects the class as a whole. Any change you make to the class affects all of its objects. Within a class definition, a preceding `@classmethod` decorator indicates that that following function is a class method. Also, the first parameter to the method is the class itself. The Python tradition is to call the parameter `cls`, because `class` is a reserved word and can't be used here. Let's define a class method for `A` that counts how many object instances have been made from it:

```
>>> class A():
...     count = 0
...     def __init__(self):
...         A.count += 1
...     def exclaim(self):
...         print("I'm an A!")
...     @classmethod
...     def kids(cls):
...         print("A has", cls.count, "little objects.")
...
>>>
>>> easy_a = A()
>>> breezy_a = A()
>>> wheezy_a = A()
>>> A.kids()
A has 3 little objects.
```

Notice that we referred to `A.count` (the class attribute) rather than `self.count` (which would be an object instance attribute). In the `kids()` method, we used `cls.count`, but we could just as well have used `A.count`.

A third type of method in a class definition affects neither the class nor its objects; it's just in there for convenience instead of floating around on its own. It's a *static method*, preceded by a `@staticmethod` decorator, with no initial `self` or `class` parameter. Here's an example that serves as a commercial for the class `CoyoteWeapon`:

```
>>> class CoyoteWeapon():
...     @staticmethod
...     def commercial():
...         print('This CoyoteWeapon has been brought to you by Acme')
...
>>>
>>> CoyoteWeapon.commercial()
This CoyoteWeapon has been brought to you by Acme
```

Notice that we didn't need to create an object from class `CoyoteWeapon` to access this method. Very class-y.

Duck Typing

Python has a loose implementation of *polymorphism*; this means that it applies the same operation to different objects, regardless of their class.

Let's use the same `__init__()` initializer for all three `Quote` classes now, but add two new functions:

- `who()` just returns the value of the saved `person` string
- `says()` returns the saved `words` string with the specific punctuation

And here they are in action:

```
>>> class Quote():
...     def __init__(self, person, words):
...         self.person = person
...         self.words = words
...     def who(self):
...         return self.person
...     def says(self):
...         return self.words + '.'
...
>>> class QuestionQuote(Quote):
...      def says(self):
...          return self.words + '?'
...
>>> class ExclamationQuote(Quote):
...      def says(self):
...          return self.words + '!'
...
>>>
```

We didn't change how `QuestionQuote` or `ExclamationQuote` were initialized, so we didn't override their `__init__()` methods. Python then automatically calls the `__init__()` method of the parent class `Quote` to store the instance variables `person` and `words`. That's why we can access `self.words` in objects created from the subclasses `QuestionQuote` and `ExclamationQuote`.

Next up, let's make some objects:

```
>>> hunter = Quote('Elmer Fudd', "I'm hunting wabbits")
>>> print(hunter.who(), 'says:', hunter.says())
Elmer Fudd says: I'm hunting wabbits.

>>> hunted1 = QuestionQuote('Bugs Bunny', "What's up, doc")
>>> print(hunted1.who(), 'says:', hunted1.says())
Bugs Bunny says: What's up, doc?

>>> hunted2 = ExclamationQuote('Daffy Duck', "It's rabbit season")
>>> print(hunted2.who(), 'says:', hunted2.says())
Daffy Duck says: It's rabbit season!
```

Three different versions of the `says()` method provide different behavior for the three classes. This is traditional polymorphism in object-oriented languages. Python goes a little further and lets you run the `who()` and `says()` methods of *any* objects that have them. Let's define a class called BabblingBrook that has no relation to our previous woodsy hunter and huntees (descendants of the Quote class):

```
>>> class BabblingBrook():
...     def who(self):
...         return 'Brook'
...     def says(self):
...         return 'Babble'
...
>>> brook = BabblingBrook()
```

Now, run the who() and says() methods of various objects, one (brook) completely unrelated to the others:

```
>>> def who_says(obj):
...     print(obj.who(), 'says', obj.says())
...
>>> who_says(hunter)
Elmer Fudd says I'm hunting wabbits.
>>> who_says(hunted1)
Bugs Bunny says What's up, doc?
>>> who_says(hunted2)
Daffy Duck says It's rabbit season!
>>> who_says(brook)
Brook says Babble
```

This behavior is sometimes called *duck typing*, after the old saying:

> If it walks like a duck and quacks like a duck, it's a duck.
>
> —A Wise Person

Special Methods

You can now create and use basic objects, but now let's go a bit deeper and do more.

When you type something such as `a = 3 + 8`, how do the integer objects with values 3 and 8 know how to implement +? Also, how does a know how to use = to get the result? You can get at these operators by using Python's *special methods* (you might also see them called *magic methods*). You don't need Gandalf to perform any magic, and they're not even complicated.

The names of these methods begin and end with double underscores (__). You've already seen one: `__init__` initializes a newly created object from its class definition and any arguments that were passed in.

Suppose that you have a simple Word class, and you want an equals() method that compares two words but ignores case. That is, a Word containing the value 'ha' would be considered equal to one containing 'HA'.

The example that follows is a first attempt, with a normal method we're calling `equals()`. `self.text` is the text string that this `Word` object contains, and the `equals()` method compares it with the text string of `word2` (another `Word` object):

```
>>> class Word():
...     def __init__(self, text):
...         self.text = text
...
...     def equals(self, word2):
...         return self.text.lower() == word2.text.lower()
...
```

Then, make three `Word` objects from three different text strings:

```
>>> first = Word('ha')
>>> second = Word('HA')
>>> third = Word('eh')
```

When strings `'ha'` and `'HA'` are compared to lowercase, they should be equal:

```
>>> first.equals(second)
True
```

But the string `'eh'` will not match `'ha'`:

```
>>> first.equals(third)
False
```

We defined the method `equals()` to do this lowercase conversion and comparison. It would be nice to just say `if first == second`, just like Python's built-in types. So, let's do that. We change the `equals()` method to the special name `__eq__()` (you'll see why in a moment):

```
>>> class Word():
...     def __init__(self, text):
...         self.text = text
...     def __eq__(self, word2):
...         return self.text.lower() == word2.text.lower()
...
```

Let's see if it works:

```
>>> first = Word('ha')
>>> second = Word('HA')
>>> third = Word('eh')
>>> first == second
True
>>> first == third
False
```

Magic! All we needed was the Python's special method name for testing equality, `__eq__()`. Tables 6-1 and 6-2 list the names of the most useful magic methods.

Table 6-1. Magic methods for comparison

`__eq__`(*self, other*)	*self* == *other*
`__ne__`(*self, other*)	*self* != *other*
`__lt__`(*self, other*)	*self* < *other*
`__gt__`(*self, other*)	*self* > *other*
`__le__`(*self, other*)	*self* <= *other*
`__ge__`(*self, other*)	*self* >= *other*

Table 6-2. Magic methods for math

`__add__`(*self, other*)	*self* + *other*
`__sub__`(*self, other*)	*self* - *other*
`__mul__`(*self, other*)	*self* * *other*
`__floordiv__`(*self, other*)	*self* // *other*
`__truediv__`(*self, other*)	*self* / *other*
`__mod__`(*self, other*)	*self* % *other*
`__pow__`(*self, other*)	*self* ** *other*

You aren't restricted to use the math operators such as `+` (magic method `__add__()`) and `-` (magic method `__sub__()`) with numbers. For instance, Python string objects use `+` for concatenation and `*` for duplication. There are many more, documented online at Special method names. The most common among them are presented in Table 6-3.

Table 6-3. Other, miscellaneous magic methods

`__str__`(*self*)	`str`(*self*)
`__repr__`(*self*)	`repr`(*self*)
`__len__`(*self*)	`len`(*self*)

Besides `__init__()`, you might find yourself using `__str__()` the most in your own methods. It's how you print your object. It's used by `print()`, `str()`, and the string formatters that you can read about in Chapter 7. The interactive interpreter uses the `__repr__()` function to echo variables to output. If you fail to define either `__str__()` or `__repr__()`, you get Python's default string version of your object:

```
>>> first = Word('ha')
>>> first
<__main__.Word object at 0x1006ba3d0>
>>> print(first)
<__main__.Word object at 0x1006ba3d0>
```

Let's add both `__str__()` and `__repr__()` methods to the `Word` class to make it prettier:

```
>>> class Word():
...     def __init__(self, text):
...         self.text = text
...     def __eq__(self, word2):
...         return self.text.lower() == word2.text.lower()
...     def __str__(self):
...         return self.text
...     def __repr__(self):
...         return 'Word("'  self.text  '")'
...
>>> first = Word('ha')
>>> first          # uses __repr__
Word("ha")
>>> print(first)   # uses __str__
ha
```

To explore even more special methods, check out the Python documentation.

Aggregation and Composition

Inheritance is a good technique to use when you want a child class to act like its parent class most of the time (when child *is-a* parent). It's tempting to build elaborate inheritance hierarchies, but sometimes *composition* or *aggregation* make more sense. What's the difference? In composition, one thing is part of another. A duck *is-a* bird (inheritance), but *has-a* tail (composition). A tail is not a kind of duck, but part of a duck. In this next example, let's make `bill` and `tail` objects and provide them to a new `duck` object:

```
>>> class Bill():
...     def __init__(self, description):
...         self.description = description
...
>>> class Tail():
...     def __init__(self, length):
...         self.length = length
```

```
...
>>> class Duck():
...     def __init__(self, bill, tail):
...         self.bill = bill
...         self.tail = tail
...     def about(self):
...         print('This duck has a', self.bill.description,
...             'bill and a', self.tail.length, 'tail')
...
>>> a_tail = Tail('long')
>>> a_bill = Bill('wide orange')
>>> duck = Duck(a_bill, a_tail)
>>> duck.about()
This duck has a wide orange bill and a long tail
```

Aggregation expresses relationships, but is a little looser: one thing *uses* another, but both exist independently. A duck *uses* a lake, but one is not a part of the other.

When to Use Classes and Objects versus Modules

Here are some guidelines for deciding whether to put your code in a class or a module:

- Objects are most useful when you need a number of individual instances that have similar behavior (methods), but differ in their internal states (attributes).
- Classes support inheritance, modules don't.
- If you want only one of something, a module might be best. No matter how many times a Python module is referenced in a program, only one copy is loaded. (Java and C++ programmers: if you're familiar with the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, you can use a Python module as a *singleton*.)
- If you have a number of variables that contain multiple values and can be passed as arguments to multiple functions, it might be better to define them as classes. For example, you might use a dictionary with keys such as `size` and `color` to represent a color image. You could create a different dictionary for each image in your program, and pass them as arguments to functions such as `scale()` or `transform()`. This can get messy as you add keys and functions. It's more coherent to define an `Image` class with attributes `size` or `color` and methods `scale()` and `transform()`. Then, all the data and methods for a color image are defined in one place.
- Use the simplest solution to the problem. A dictionary, list, or tuple is simpler, smaller, and faster than a module, which is usually simpler than a class.

Guido's advice:

> Avoid overengineering datastructures. Tuples are better than objects (try namedtuple too though). Prefer simple fields over getter/setter functions … Built-in datatypes are your friends. Use more numbers, strings, tuples, lists, sets, dicts. Also check out the collections library, esp. deque.
>
> —Guido van Rossum

Named Tuples

Because Guido just mentioned them and I haven't yet, this is a good place to talk about *named tuples*. A named tuple is a subclass of tuples with which you can access values by name (with `.name`) as well as by position (with `[ offset ]`).

Let's take the example from the previous section and convert the `Duck` class to a named tuple, with `bill` and `tail` as simple string attributes. We'll call the `namedtuple` function with two arguments:

- The name
- A string of the field names, separated by spaces

Named tuples are not automatically supplied with Python, so you need to load a module before using them. We do that in the first line of the following example:

```
>>> from collections import namedtuple
>>> Duck = namedtuple('Duck', 'bill tail')
>>> duck = Duck('wide orange', 'long')
>>> duck
Duck(bill='wide orange', tail='long')
>>> duck.bill
'wide orange'
>>> duck.tail
'long'
```

You can also make a named tuple from a dictionary:

```
>>> parts = {'bill': 'wide orange', 'tail': 'long'}
>>> duck2 = Duck(**parts)
>>> duck2
Duck(bill='wide orange', tail='long')
```

In the preceding code, take a look at `**parts`. This is a *keyword argument*. It extracts the keys and values from the `parts` dictionary and supplies them as arguments to `Duck()`. It has the same effect as:

```
>>> duck2 = Duck(bill = 'wide orange', tail = 'long')
```

Named tuples are immutable, but you can replace one or more fields and return another named tuple:

```
>>> duck3 = duck2._replace(tail='magnificent', bill='crushing')
>>> duck3
Duck(bill='crushing', tail='magnificent')
```

We could have defined `duck` as a dictionary:

```
>>> duck_dict = {'bill': 'wide orange', 'tail': 'long'}
>>> duck_dict
{'tail': 'long', 'bill': 'wide orange'}
```

You can add fields to a dictionary:

```
>>> duck_dict['color'] = 'green'
>>> duck_dict
{'color': 'green', 'tail': 'long', 'bill': 'wide orange'}
```

But not to a named tuple:

```
>>> duck.color = 'green'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object has no attribute 'color'
```

To recap, here are some of the pros of a named tuple:

- It looks and acts like an immutable object.
- It is more space- and time-efficient than objects.
- You can access attributes by using dot notation instead of dictionary-style square brackets.
- You can use it as a dictionary key.

Things to Do

6.1. Make a class called `Thing` with no contents and print it. Then, create an object called `example` from this class and also print it. Are the printed values the same or different?

6.2. Make a new class called `Thing2` and assign the value `'abc'` to a class attribute called `letters`. Print `letters`.

6.3. Make yet another class called, of course, `Thing3`. This time, assign the value `'xyz'` to an instance (object) attribute called `letters`. Print `letters`. Do you need to make an object from the class to do this?

6.4. Make a class called `Element`, with instance attributes `name`, `symbol`, and `number`. Create an object of this class with the values `'Hydrogen'`, `'H'`, and `1`.

6.5. Make a dictionary with these keys and values: `'name': 'Hydrogen', 'symbol': 'H', 'number': 1`. Then, create an object called `hydrogen` from class `Element` using this dictionary.

6.6. For the `Element` class, define a method called `dump()` that prints the values of the object's attributes (`name`, `symbol`, and `number`). Create the `hydrogen` object from this new definition and use `dump()` to print its attributes.

6.7. Call `print(hydrogen)`. In the definition of `Element`, change the name of method `dump` to `__str__`, create a new `hydrogen` object, and call `print(hydrogen)` again.

6.8. Modify `Element` to make the attributes `name`, `symbol`, and `number` private. Define a getter property for each to return its value.

6.9. Define three classes: `Bear`, `Rabbit`, and `Octothorpe`. For each, define only one method: `eats()`. This should return `'berries'` (`Bear`), `'clover'` (`Rabbit`), or `'campers'` (`Octothorpe`). Create one object from each and print what it eats.

6.10. Define these classes: `Laser`, `Claw`, and `SmartPhone`. Each has only one method: `does()`. This returns `'disintegrate'` (`Laser`), `'crush'` (`Claw`), or `'ring'` (`SmartPhone`). Then, define the class `Robot` that has one instance (object) of each of these. Define a `does()` method for the `Robot` that prints what its component objects do.