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
```