CHAPTER 1
The Python Data Model

> Guido's sense of the aesthetics of language design is amazing. I've met many fine language designers who could build theoretically beautiful languages that no one would ever use, but Guido is one of those rare people who can build a language that is just slightly less theoretically beautiful but thereby is a joy to write programs in[1].
>
> — Jim Hugunin
> *creator of Jython, co-creator of AspectJ, architect of the .Net DLR*

One of the best qualities of Python is its consistency. After working with Python for a while, you are able to start making informed, correct guesses about features that are new to you.

However, if you learned another object oriented language before Python, you may have found it strange to spell `len(collection)` instead of `collection.len()`. This apparent oddity is the tip of an iceberg which, when properly understood, is the key to everything we call *Pythonic*. The iceberg is called the Python Data Model, and it describes the API that you can use to make your own objects play well with the most idiomatic language features.

You can think of the Data Model as a description of Python as a framework. It formalizes the interfaces of the building blocks of the language itself, such as sequences, iterators, functions, classes, context managers and so on.

While coding with any framework, you spend a lot of time implementing methods that are called by the framework. The same happens when you leverage the Python Data Model. The Python interpreter invokes special methods to perform basic object operations, often triggered by special syntax. The special method names are always spelled with leading and trailing double underscores, i.e. `__getitem__`. For example, the syntax

1. Story of Jython, written as a foreword to *Jython Essentials* (O'Reilly, 2002), by Samuele Pedroni and Noel Rappin.

`obj[key]` is supported by the `__getitem__` special method. To evaluate `my_collection[key]`, the interpreter calls `my_collection.__getitem__(key)`.

The special method names allow your objects to implement, support and interact with basic language constructs such as:

- iteration;
- collections;
- attribute access;
- operator overloading;
- function and method invocation;
- object creation and destruction;
- string representation and formatting;
- managed contexts (i.e. **`with`** blocks);

Magic and dunder

The term *magic method* is slang for special method, but when talking about a specific method like `__getitem__`, some Python developers take the shortcut of saying "under-under-getitem" which is ambiguous, since the syntax `__x` has another special meaning[2]. But being precise and pronouncing "under-under-getitem-under-under" is tiresome, so I follow the lead of author and teacher Steve Holden and say "dunder-getitem". All experienced Pythonistas understand that shortcut. As a result, the special methods are also known as *dunder methods* [3].

A Pythonic Card Deck

The following is a very simple example, but it demonstrates the power of implementing just two special methods, `__getitem__` and `__len__`.

Example 1-1 is a class to represent a deck of playing cards:

2. See "Private and "protected" attributes in Python" on page 263
3. I personally first heard "dunder" from Steve Holden. The English language Wikipedia credits Mark Johnson and Tim Hochberg for the first written records of "dunder" in responses to the question "How do you pronounce __ (double underscore)?" in the python-list in September 26, 2002: Johnson's message; Hochberg's (11 minutes later).

Example 1-1. A deck as a sequence of cards.

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

The first thing to note is the use of `collections.namedtuple` to construct a simple class to represent individual cards. Since Python 2.6, `namedtuple` can be used to build classes of objects that are just bundles of attributes with no custom methods, like a database record. In the example we use it to provide a nice representation for the cards in the deck, as shown in the console session:

```
>>> beer_card = Card('7', 'diamonds')
>>> beer_card
Card(rank='7', suit='diamonds')
```

But the point of this example is the `FrenchDeck` class. It's short, but it packs a punch. First, like any standard Python collection, a deck responds to the `len()` function by returning the number of cards in it.

```
>>> deck = FrenchDeck()
>>> len(deck)
52
```

Reading specific cards from the deck, say, the first or the last, should be as easy as `deck[0]` or `deck[-1]`, and this is what the `__getitem__` method provides.

```
>>> deck[0]
Card(rank='2', suit='spades')
>>> deck[-1]
Card(rank='A', suit='hearts')
```

Should we create a method to pick a random card? No need. Python already has a function to get a random item from a sequence: `random.choice`. We can just use it on a deck instance:

```
>>> from random import choice
>>> choice(deck)
```

```
Card(rank='3', suit='hearts')
>>> choice(deck)
Card(rank='K', suit='spades')
>>> choice(deck)
Card(rank='2', suit='clubs')
```

We've just seen two advantages of using special methods to leverage the Python Data Model:

1. The users of your classes don't have to memorize arbitrary method names for standard operations ("How to get the number of items? Is it .size() .length() or what?")
2. It's easier to benefit from the rich Python standard library and avoid reinventing the wheel, like the `random.choice` function.

But it gets better.

Because our `__getitem__` delegates to the `[]` operator of `self._cards`, our deck automatically supports slicing. Here's how we look at the top three cards from a brand new deck, and then pick just the aces by starting on index 12 and skipping 13 cards at a time:

```
>>> deck[:3]
[Card(rank='2', suit='spades'), Card(rank='3', suit='spades'),
Card(rank='4', suit='spades')]
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),
Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

Just by implementing the `__getitem__` special method, our deck is also iterable:

```
>>> for card in deck:  # doctest: +ELLIPSIS
...   print(card)
Card(rank='2', suit='spades')
Card(rank='3', suit='spades')
Card(rank='4', suit='spades')
...
```

The deck can also be iterated in reverse:

```
>>> for card in reversed(deck):  # doctest: +ELLIPSIS
...   print(card)
Card(rank='A', suit='hearts')
Card(rank='K', suit='hearts')
Card(rank='Q', suit='hearts')
...
```

Ellipsis in doctests

Whenever possible, the Python console listings in this book were extracted from doctests to insure accuracy. When the output was too long, the elided part is marked by an ellipsis `...` like in the last line above. In such cases, we used the `# doctest: +ELLIPSIS` directive to make the doctest pass. If you are trying these examples in the interactive console, you may omit the doctest directives altogether.

Iteration is often implicit. If a collection has no `__contains__` method, the `in` operator does a sequential scan. Case in point: `in` works with our `FrenchDeck` class because it is iterable. Check it out:

```
>>> Card('Q', 'hearts') in deck
True
>>> Card('7', 'beasts') in deck
False
```

How about sorting? A common system of ranking cards is by rank (with aces being highest), then by suit in the order: spades (highest), then hearts, diamonds and clubs (lowest). Here is a function that ranks cards by that rule, returning `0` for the 2 of clubs and `51` for the ace of spades:

```
suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0)

def spades_high(card):
    rank_value = FrenchDeck.ranks.index(card.rank)
    return rank_value * len(suit_values) + suit_values[card.suit]
```

Given `spades_high`, we can now list our deck in order of increasing rank:

```
>>> for card in sorted(deck, key=spades_high):  # doctest: +ELLIPSIS
...     print(card)
Card(rank='2', suit='clubs')
Card(rank='2', suit='diamonds')
Card(rank='2', suit='hearts')
... (46 cards ommitted)
Card(rank='A', suit='diamonds')
Card(rank='A', suit='hearts')
Card(rank='A', suit='spades')
```

Although FrenchDeck implicitly inherits from `object`[4], its functionality is not inherited, but comes from leveraging the Data Model and composition. By implementing the special methods `__len__` and `__getitem__` our `FrenchDeck` behaves like a standard Python sequence, allowing it to benefit from core language features — like iteration and slicing — and from the standard library, as shown by the examples using `ran`

4. In Python 2 you'd have to be explicit and write `FrenchDeck(object)`, but that's the default in Python 3.

`dom.choice`, `reversed` and `sorted`. Thanks to composition, the `__len__` and `__getitem__` implementations can hand off all the work to a `list` object, `self._cards`.

How about shuffling?

As implemented so far, a `FrenchDeck` cannot be shuffled, because it is *immutable*: the cards and their positions cannot be changed, except by violating encapsulation and handling the `_cards` attribute directly. In Chapter 11 that will be fixed by adding a one-line `__setitem__` method.

How special methods are used

The first thing to know about special methods is that they are meant to be called by the Python interpreter, and not by you. You don't write `my_object.__len__()`. You write `len(my_object)` and, if `my_object` is an instance of a user defined class, then Python calls the `__len__` instance method you implemented.

But for built-in types like `list`, `str`, `bytearray` etc., the interpreter takes a shortcut: the CPython implementation of `len()` actually returns the value of the `ob_size` field in the `PyVarObject` C struct that represents any variable-sized built-in object in memory. This is much faster than calling a method.

More often than not, the special method call is implicit. For example, the statement `for i in x:` actually causes the invocation of `iter(x)` which in turn may call `x.__iter__()` if that is available.

Normally, your code should not have many direct calls to special methods. Unless you are doing a lot of metaprogramming, you should be implementing special methods more often than invoking them explicitly. The only special method that is frequently called by user code directly is `__init__`, to invoke the initializer of the superclass in your own `__init__` implementation.

If you need to invoke a special method, it is usually better to call the related built-in function, such as `len`, `iter`, `str` etc. These built-ins call the corresponding special method, but often provide other services and — for built-in types — are faster than method calls. See for example "A closer look at the `iter` function" on page 438 in Chapter 14.

Avoid creating arbitrary, custom attributes with the `__foo__` syntax because such names may acquire special meanings in the future, even if they are unused today.

Emulating numeric types

Several special methods allow user objects to respond to operators such as +. We will cover that in more detail in Chapter 13, but here our goal is to further illustrate the use of special methods through another simple example.

We will implement a class to represent 2-dimensional vectors, i.e. Euclidean vectors like those used in math and physics (see Figure 1-1).

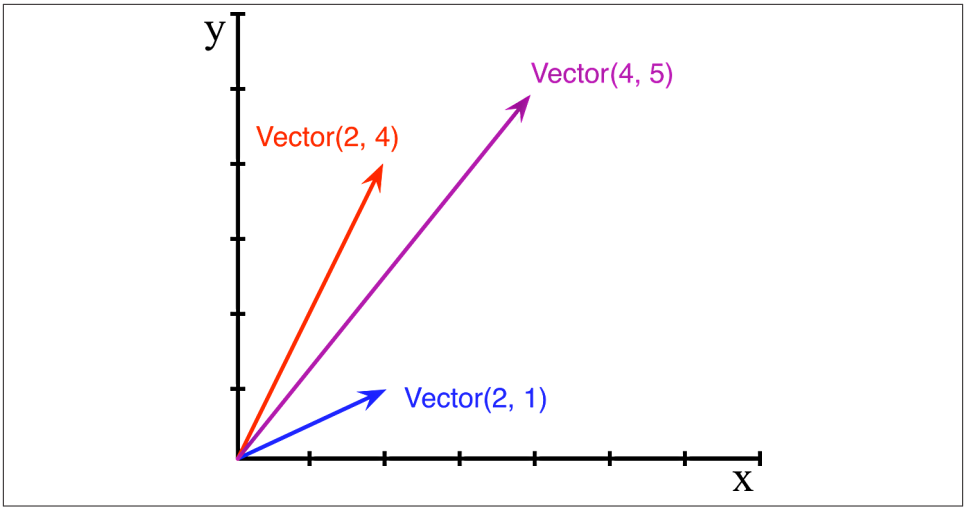

Figure 1-1. Example of 2D vector addition. `Vector(2, 4) + Vector(2, 1)` results in `Vector(4, 5)`.

The built-in `complex` type can be used to represent 2D vectors, but our class can be extended to represent n-dimensional vectors. We will do that in Chapter 14.

We will start by designing the API for such a class by writing a simulated console session which we can use later as doctest. The following snippet tests the vector addition pictured in Figure 1-1:

```
>>> v1 = Vector(2, 4)
>>> v2 = Vector(2, 1)
>>> v1 + v2
Vector(4, 5)
```

Note how the + operator produces a `Vector` result which is displayed in a friendly manner in the console.

The `abs` built-in function returns the absolute value of integers and floats, and the magnitude of `complex` numbers, so to be consistent our API also uses `abs` to calculate the magnitude of a vector:

```
>>> v = Vector(3, 4)
>>> abs(v)
5.0
```

We can also implement the `*` operator to perform scalar multiplication, i.e. multiplying a vector by a number to produce a new vector with the same direction and a multiplied magnitude:

```
>>> v * 3
Vector(9, 12)
>>> abs(v * 3)
15.0
```

Example 1-2 is a `Vector` class implementing the operations just described, through the use of the special methods `__repr__`, `__abs__`, `__add__` and `__mul__`:

Example 1-2. A simple 2D vector class.

```
from math import hypot

class Vector:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Vector(%r, %r)' % (self.x, self.y)

    def __abs__(self):
        return hypot(self.x, self.y)

    def __bool__(self):
        return bool(abs(self))

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)
```

Note that although we implemented four special methods (apart from `__init__`), none of them is directly called within the class or in the typical usage of the class illustrated by the console listings. As mentioned before, the Python interpreter is the only frequent

caller of most special methods. In the next sections we discuss the code for each special method.

String representation

The `__repr__` special method is called by the `repr` built-in to get string representation of the object for inspection. If we did not implement `__repr__`, vector instances would be shown in the console like `<Vector object at 0x10e100070>`.

The interactive console and debugger call `repr` on the results of the expressions evaluated, as does the `'%r'` place holder in classic formatting with `%` operator, and the `!r` conversion field in the new Format String Syntax used in the `str.format` method[5].

Note that in our `__repr__` implementation we used `%r` to obtain the standard representation of the attributes to be displayed. This is good practice, as it shows the crucial difference between `Vector(1, 2)` and `Vector('1', '2')` — the latter would not work in the context of this example, because the constructors arguments must be numbers, not `str`.

The string returned by `__repr__` should be unambiguous and, if possible, match the source code necessary to recreate the object being represented. That is why our chosen representation looks like calling the constructor of the class, e.g. `Vector(3, 4)`.

Contrast `__repr__` with with `__str__`, which is called by the `str()` constructor and implicitly used by the `print` function. `__str__` should return a string suitable for display to end-users.

If you only implement one of these special methods, choose `__repr__`, because when no custom `__str__` is available, Python will call `__repr__` as a fallback.

Difference between `__str__` and `__repr__` in Python is a StackOverflow question with excellent contributions from Pythonistas Alex Martelli and Martijn Pieters.

Arithmetic operators

Example 1-2 implements two operators: `+` and `*`, to show basic usage of `__add__` and `__mul__`. Note that in both cases, the methods create and return a new instance of

5. Speaking of the `%` operator and the `str.format` method, the reader will notice I use both in this book, as does the Python community at large. I am increasingly favoring the more powerful `str.format`, but I am aware many Pythonistas prefer the simpler `%`, so we'll probably see both in Python source code for the foreseeable future.

`Vector`, and do not modify either operand — `self` or `other` are merely read. This is the expected behavior of infix operators: to create new objects and not touch their operands. I will have a lot more to say about that in Chapter 13.

As implemented, Example 1-2 allows multiplying a `Vector` by a number, but not a number by a `Vector`, which violates the commutative property of multiplication. We will fix that with the special method `__rmul__` in Chapter 13.

Boolean value of a custom type

Although Python has a `bool` type, it accepts any object in a boolean context, such as the expression controlling an `if` or `while` statement, or as operands to `and`, `or` and `not`. To determine whether a value `x` is *truthy* or *falsy*, Python applies `bool(x)`, which always returns `True` or `False`.

By default, instances of user-defined classes are considered truthy, unless either `__bool__` or `__len__` is implemented. Basically, `bool(x)` calls `x.__bool__()` and uses the result. If `__bool__` is not implemented, Python tries to invoke `x.__len__()`, and if that returns zero, `bool` returns `False`. Otherwise `bool` returns `True`.

Our implementation of `__bool__` is conceptually simple: it returns `False` if the magnitude of the vector is zero, `True` otherwise. We convert the magnitude to a boolean using `bool(abs(self))` because `__bool__` is expected to return a boolean.

Note how the special method `__bool__` allows your objects to be consistent with the truth value testing rules defined in the Built-in Types chapter of the Python Standard Library documentation.

A faster implementation of `Vector.__bool__` is this:

```
def __bool__(self):
    return bool(self.x or self.y)
```

This is harder to read, but avoids the trip through `abs`, `__abs__`, the squares and square root. The explicit conversion to `bool` is needed because `__bool__` must return a boolean and `or` returns either operand as is: `x or y` evaluates to `x` if that is *truthy*, otherwise the result is `y`, whatever that is.

Overview of special methods

The Data Model page of the Python Language Reference lists 83 special method names, 47 of which are used to implement arithmetic, bitwise and comparison operators.

As an overview of what is available, see Table 1-1 and Table 1-2.

The grouping shown in the following tables is not exactly the same as in the official documentation.

Table 1-1. Special method names (operators excluded).

category	method names
string/bytes representation	`__repr__`, `__str__`, `__format__`, `__bytes__`
conversion to number	`__abs__`, `__bool__`, `__complex__`, `__int__`, `__float__`, `__hash__`, `__index__`
emulating collections	`__len__`, `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`
iteration	`__iter__`, `__reversed__`, `__next__`
emulating callables	`__call__`
context management	`__enter__`, `__exit__`
instance creation and destruction	`__new__`, `__init__`, `__del__`
attribute management	`__getattr__`, `__getattribute__`, `__setattr__`, `__delattr__`, `__dir__`
attribute descriptors	`__get__`, `__set__`, `__delete__`
class services	`__prepare__`, `__instancecheck__`, `__subclasscheck__`

Table 1-2. Special method names for operators.

category	method names and related operators
unary numeric operators	`__neg__` -, `__pos__` +, `__abs__` `abs()`
rich compartison operators	`__lt__` >, `__le__` <=, `__eq__` ==, `__ne__` !=, `__gt__` >, `__ge__` >=
arithmetic operators	`__add__` +, `__sub__` -, `__mul__` *, `__truediv__` /, `__floordiv__` //, `__mod__` %, `__divmod__` `divmod()`, `__pow__` ** or `pow()`, `__round__` `round()`
reversed arithmetic operators	`__radd__`, `__rsub__`, `__rmul__`, `__rtruediv__`, `__rfloordiv__`, `__rmod__`, `__rdivmod__`, `__rpow__`
augmented assignment arithmetic operators	`__iadd__`, `__isub__`, `__imul__`, `__itruediv__`, `__ifloordiv__`, `__imod__`, `__ipow__`
bitwise operators	`__invert__` ~, `__lshift__` <<, `__rshift__` >>, `__and__` &, `__or__` |, `__xor__` ^
reversed bitwise operators	`__rlshift__`, `__rrshift__`, `__rand__`, `__rxor__`, `__ror__`
augmented assignment bitwise operators	`__ilshift__`, `__irshift__`, `__iand__`, `__ixor__`, `__ior__`

The reversed operators are fallbacks used when operands are swapped (`b * a` instead of `a * b`), while augmented assignment are shortcuts combining an infix operator with variable assignment (`a = a * b` becomes `a *= b`). Chapter 13 explains both reversed operators and augmented assignment in detail.

Why `len` is not a method

I asked this question to core developer Raymond Hettinger in 2013 and the key to his answer was a quote from the Zen of Python: "practicality beats purity". In "How special methods are used" on page 8 I described how `len(x)` runs very fast when `x` is an instance of a built-in type. No method is called for the built-in objects of CPython: the length is simply read from a field in a C struct. Getting the number of items in a collection is a common operation and must work efficiently for such basic and diverse types as `str`, `list`, `memoryview` etc.

In other words, `len` is not called as a method because it gets special treatment as part of the Python Data Model, just like `abs`. But thanks to the special method `__len__` you can also make `len` work with your own custom objects. This is fair compromise between the need for efficient built-in objects and the consistency of the language. Also from the Zen of Python: "Special cases aren't special enough to break the rules."

If you think of `abs` and `len` as unary operators you may be more inclined to forgive their functional look-and-feel, as opposed to the method call syntax one might expect in a OO language. In fact, the ABC language — a direct ancestor of Python which pioneered many of its features — had an `#` operator that was the equivalent of `len` (you'd write `#s`). When used as an infix operator, written `x#s`, it counted the occurrences of `x` in `s`, which in Python you get as `s.count(x)`, for any sequence `s`.

Chapter summary

By implementing special methods, your objects can behave like the built-in types, enabling the expressive coding style the community considers Pythonic.

A basic requirement for a Python object is to provide usable string representations of itself, one used for debugging and logging, another for presentation to end users. That is why the special methods `__repr__` and `__str__` exist in the Data Model.

Emulating sequences, as shown with the `FrenchDeck` example, is one of the most widely used applications of the special methods. Making the most of sequence types is the

subject of Chapter 2, and implementing your own sequence will be covered in Chapter 10 we will create a multi-dimensional extension of the `Vector` class.

Thanks to operator overloading, Python offers a rich selection of numeric types, from the built-ins to `decimal.Decimal` and `fractions.Fraction`, all supporting infix arithmetic operators. Implementing operators, including reversed operators and augmented assignment will be shown in Chapter 13 via enhancements of the `Vector` example.

The use and implementation of the majority of the remaining special methods of the Python Data Model is covered throughout this book.

Further reading

The Data Model chapter of the Python Language Reference is the canonical source for the subject of this chapter and much of this book.

Python in a Nutshell, 2nd Edition, by Alex Martelli, has excellent coverage of the Data Model. As I write this, the most recent edition of the *Nutshell* book is from 2006 and focuses on Python 2.5, but there were very few changes in the Data Model since then, and Martelli's description of the mechanics of attribute access is the most authoritative I've seen apart from the actual C source code of CPython. Martelli is also a prolific contributor to Stack Overflow, with more than 5000 answers posted. See his user profile at *http://stackoverflow.com/users/95810/alex-martelli*.

David Beazley has two books covering the Data Model in detail in the context of Python 3: *Python Essential Reference, 4th Edition*, and *Python Cookbook, 3rd Edition*, co-authored with Brian K. Jones.

The Art of the Metaobject Protocol (AMOP), by Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow explains the concept of a MOP (Meta Object Protocol), of which the Python Data Model is one example.

Soapbox

Data Model or Object Model?

What the Python documentation calls the "Python Data Model", most authors would say is the "Python Object Model". Alex Martelli's *Python in a Nutshell 2e* and David Beazley's *Python Essential Reference 4e* are the best books covering the "Python Data Model", but they always refer to it as the "object model". In the English language Wikipedia, the first definition of Object Model is "The properties of objects in general in a specific computer programming language." This is what the "Python Data Model" is about. In this book I will use "Data Model" because that is how the Python object model is called in the documentation, and is the title of the chapter of the Python Language Reference most relevant to our discussions.

Magic methods

The Ruby community calls their equivalent of the special methods *magic methods*. Many in the Python community adopt that term as well. I believe the special methods are actually the opposite of magic. Python and Ruby are the same in this regard: both empower their users with a rich metaobject protocol which is not magic, but enables users to leverage the same tools available to core developers.

In contrast, consider JavaScript. Objects in that language have features that are magic, in the sense that you cannot emulate them in your own user-defined objects. For example, before JavaScript 1.8.5 you could not define read-only attributes in your JavaScript objects, but some built-in objects always had read-only attributes. In JavaScript, read-only attributes were "magic", requiring supernatural powers that a user of the language did not have until ECMAScript 5.1 came out in 2009. The metaobject protocol of JavaScript is evolving, but historically it has been more limited than those of Python and Ruby.

Metaobjects

The Art of The Metaobject Procotol (AMOP) is my favorite computer book title. Less subjectively, the term *metaobject protocol* is useful to think about the Python Data Model and similar features in other languages. The *metaobject* part refers to the objects that are the building blocks of the language itself. In this context, *prococol* is a synonym of *interface*. So a *metaobject protocol* is a fancy synonym for object model: an API for core language constructs.

A rich metaobject protocol enables extending a language to support new programming paradigms. Gregor Kiczales, the first author of the AMOP book, later became a pioneer in aspect-oriented programming and the initial author of AspectJ, an extension of Java implementing that paradigm. Aspect-oriented programming is much easier to implement in a dynamic language like Python, and several frameworks do it, but the most important is `zope.interface`, which is briefly discussed in the **Further reading** section of Chapter 11.