CHAPTER 11
Interfaces: From Protocols to ABCs

> An abstract class represents an interface.[1]
>
> — Bjarne Stroustrup
> *Creator of C++*

Interfaces are the subject of this chapter: from the dynamic protocols that are the hallmark of *duck typing* to abstract base classes (ABCs) that make interfaces explicit and verify implementations for conformance.

If you have a Java, C#, or similar background, the novelty here is in the informal protocols of duck typing. But for the long-time Pythonista or Rubyist, that is the "normal" way of thinking about interfaces, and the news is the formality and type-checking of ABCs. The language was 15 years old when ABCs were introduced in Python 2.6.

We'll start the chapter by reviewing how the Python community traditionally understood interfaces as somewhat loose—in the sense that a partially implemented interface is often acceptable. We'll make that clear through a couple examples that highlight the dynamic nature of duck typing.

Then, a guest essay by Alex Martelli will introduce ABCs and give name to a new trend in Python programming. The rest of the chapter will be devoted to ABCs, starting with their common use as superclasses when you need to implement an interface. We'll then see when an ABC checks concrete subclasses for conformance to the interface it defines, and how a registration mechanism lets developers declare that a class implements an interface without subclassing. Finally, we'll see how an ABC can be programmed to automatically "recognize" arbitrary classes that conform to its interface—without subclassing or explicit registration.

1. Bjarne Stroustrup, *The Design and Evolution of C++* (Addison-Wesley, 1994), p. 278.

We will implement a new ABC to see how that works, but Alex Martelli and I don't want to encourage you to start writing your own ABCs left and right. The risk of over-engineering with ABCs is very high.

ABCs, like descriptors and metaclasses, are tools for building frameworks. Therefore, only a very small minority of Python developers can create ABCs without imposing unreasonable limitations and needless work on fellow programmers.

Let's get started with the Pythonic view of interfaces.

Interfaces and Protocols in Python Culture

Python was already highly successful before ABCs were introduced, and most existing code does not use them at all. Since Chapter 1, we've been talking about *duck typing* and protocols. In "Protocols and Duck Typing" on page 279, protocols are defined as the informal interfaces that make polymorphism work in languages with dynamic typing like Python.

How do interfaces work in a dynamic-typed language? First, the basics: even without an `interface` keyword in the language, and regardless of ABCs, every class has an interface: the set public attributes (methods or data attributes) implemented or inherited by the class. This includes special methods, like `__getitem__` or `__add__`.

By definition, protected and private attributes are not part of an interface, even if "protected" is merely a naming convention (the single leading underscore) and private attributes are easily accessed (recall "Private and "Protected" Attributes in Python" on page 262). It is bad form to violate these conventions.

On the other hand, it's not a sin to have public data attributes as part of the interface of an object, because—if necessary—a data attribute can always be turned into a property implementing getter/setter logic without breaking client code that uses the plain `obj.attr` syntax. We did that in the `Vector2d` class: in Example 11-1, we see the first implementation with public `x` and `y` attributes.

Example 11-1. vector2d_v0.py: x and y are public data attributes (same code as Example 9-2)

```
class Vector2d:
    typecode = 'd'

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

    def __iter__(self):
```

```
        return (i for i in (self.x, self.y))

    # more methods follow (omitted in this listing)
```

In Example 9-7, we turned `x` and `y` into read-only properties (Example 11-2). This is a significant refactoring, but an essential part of the interface of `Vector2d` is unchanged: users can still read `my_vector.x` and `my_vector.y`.

Example 11-2. vector2d_v3.py: x and y reimplemented as properties (see full listing in Example 9-9)

```
class Vector2d:
    typecode = 'd'

    def __init__(self, x, y):
        self.__x = float(x)
        self.__y = float(y)

    @property
    def x(self):
        return self.__x

    @property
    def y(self):
        return self.__y

    def __iter__(self):
        return (i for i in (self.x, self.y))

    # more methods follow (omitted in this listing)
```

A useful complementary definition of interface is: the subset of an object's public methods that enable it to play a specific role in the system. That's what is implied when the Python documentation mentions "a file-like object" or "an iterable," without specifying a class. An interface seen as a set of methods to fulfill a role is what Smalltalkers called a *procotol*, and the term spread to other dynamic language communities. Protocols are independent of inheritance. A class may implement several protocols, enabling its instances to fulfill several roles.

Protocols are interfaces, but because they are informal—defined only by documentation and conventions—protocols cannot be enforced like formal interfaces can (we'll see how ABCs enforce interface conformance later in this chapter). A protocol may be partially implemented in a particular class, and that's OK. Sometimes all a specific API requires from "a file-like object" is that it has a `.read()` method that returns bytes. The remaining file methods may or may not be relevant in the context.

As I write this, the Python 3 documentation of `memoryview` says that it works with objects that "support the buffer protocol, which is only documented at the C API level. The `bytearray` constructor accepts an "an object conforming to the buffer interface." Now

there is a move to adopt "bytes-like object" as a friendlier term.[2] I point this out to emphasize that "X-like object," "X protocol," and "X interface" are synonyms in the minds of Pythonistas.

One of the most fundamental interfaces in Python is the sequence protocol. The interpreter goes out of its way to handle objects that provide even a minimal implementation of that protocol, as the next section demonstrates.

Python Digs Sequences

The philosophy of the Python data model is to cooperate with essential protocols as much as possible. When it comes to sequences, Python tries hard to work with even the simplest implementations.

Figure 11-1 shows how the formal `Sequence` interface is defined as an ABC.

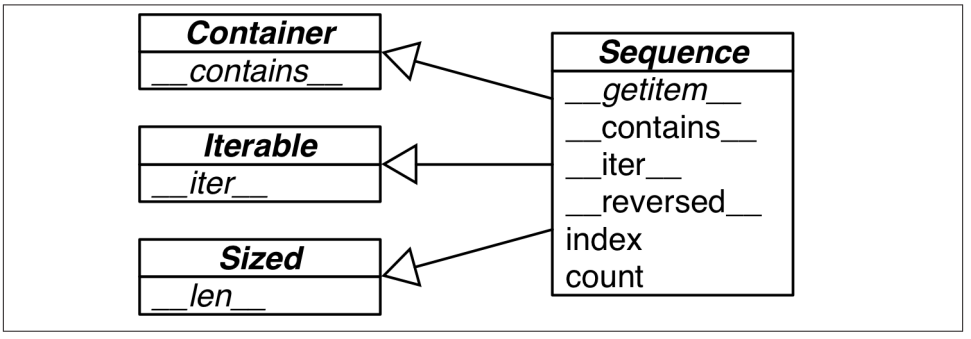

Figure 11-1. UML class diagram for the Sequence ABC and related abstract classes from collections.abc. Inheritance arrows point from subclass to its superclasses. Names in italic are abstract methods.

Now, take a look at the `Foo` class in Example 11-3. It does not inherit from `abc.Sequence`, and it only implements one method of the sequence protocol: `__getitem__` (`__len__` is missing).

Example 11-3. Partial sequence protocol implementation with __getitem__: enough for item access, iteration, and the in operator

```
>>> class Foo:
...     def __getitem__(self, pos):
...         return range(0, 30, 10)[pos]
```

2. Issue16518: "add *buffer protocol* to glossary" was actually resolved by replacing many mentions of "object that supports the buffer protocol/interface/API" with "bytes-like object"; a follow-up issue is "Other mentions of the buffer protocol".

```
...
>>> f[1]
10
>>> f = Foo()
>>> for i in f: print(i)
...
0
10
20
>>> 20 in f
True
>>> 15 in f
False
```

There is no method `__iter__` yet `Foo` instances are iterable because—as a fallback—when Python sees a `__getitem__` method, it tries to iterate over the object by calling that method with integer indexes starting with `0`. Because Python is smart enough to iterate over `Foo` instances, it can also make the `in` operator work even if `Foo` has no `__contains__` method: it does a full scan to check if an item is present.

In summary, given the importance of the sequence protocol, in the absence `__iter__` and `__contains__` Python still manages to make iteration and the `in` operator work by invoking `__getitem__`.

Our original `FrenchDeck` from Chapter 1 does not subclass from `abc.Sequence` either, but it does implement both methods of the sequence protocol: `__getitem__` and `__len__`. See Example 11-4.

Example 11-4. A deck as a sequence of cards (same as Example 1-1)

```
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]
```

A good part of the demos in Chapter 1 work because of the special treatment Python gives to anything vaguely resembling a sequence. Iteration in Python represents an

extreme form of duck typing: the interpreter tries two different methods to iterate over objects.

Now let's study another example emphasizing the dynamic nature of protocols.

Monkey-Patching to Implement a Protocol at Runtime

The `FrenchDeck` class from Example 11-4 has a major flaw: it cannot be shuffled. Years ago when I first wrote the `FrenchDeck` example I did implement a `shuffle` method. Later I had a Pythonic insight: if a `FrenchDeck` acts like a sequence, then it doesn't need its own `shuffle` method because there is already `random.shuffle`, documented as "Shuffle the sequence *x* in place."

When you follow established protocols, you improve your chances of leveraging existing standard library and third-party code, thanks to duck typing.

The standard `random.shuffle` function is used like this:

```
>>> from random import shuffle
>>> l = list(range(10))
>>> shuffle(l)
>>> l
[5, 2, 9, 7, 8, 3, 1, 4, 0, 6]
```

However, if we try to shuffle a `FrenchDeck` instance, we get an exception, as in Example 11-5.

Example 11-5. random.shuffle cannot handle FrenchDeck

```
>>> from random import shuffle
>>> from frenchdeck import FrenchDeck
>>> deck = FrenchDeck()
>>> shuffle(deck)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../python3.3/random.py", line 265, in shuffle
    x[i], x[j] = x[j], x[i]
TypeError: 'FrenchDeck' object does not support item assignment
```

The error message is quite clear: "`'FrenchDeck'` object does not support item assignment." The problem is that `shuffle` operates by swapping items inside the collection, and `FrenchDeck` only implements the *immutable* sequence protocol. Mutable sequences must also provide a `__setitem__` method.

Because Python is dynamic, we can fix this at runtime, even at the interactive console. Example 11-6 shows how to do it.

Example 11-6. Monkey patching FrenchDeck to make it mutable and compatible with random.shuffle (continuing from Example 11-5)

```
>>> def set_card(deck, position, card):  ❶
...     deck._cards[position] = card
...
>>> FrenchDeck.__setitem__ = set_card  ❷
>>> shuffle(deck)  ❸
>>> deck[:5]
[Card(rank='3', suit='hearts'), Card(rank='4', suit='diamonds'), Card(rank='4',
suit='clubs'), Card(rank='7', suit='hearts'), Card(rank='9', suit='spades')]
```

❶ Create a function that takes `deck`, `position`, and `card` as arguments.

❷ Assign that function to an attribute named `__setitem__` in the `FrenchDeck` class.

❸ `deck` can now be sorted because `FrenchDeck` now implements the necessary method of the mutable sequence protocol.

The signature of the `__setitem__` special method is defined in *The Python Language Reference* in "3.3.6. Emulating container types". Here we named the arguments `deck, position, card`—and not `self, key, value` as in the language reference—to show that every Python method starts life as a plain function, and naming the first argument `self` is merely a convention. This is OK in a console session, but in a Python source file it's much better to use `self`, `key`, and `value` as documented.

The trick is that `set_card` knows that the `deck` object has an attribute named `_cards`, and `_cards` must be a mutable sequence. The `set_card` function is then attached to the `FrenchDeck` class as the `__setitem__` special method. This is an example of *monkey patching*: changing a class or module at runtime, without touching the source code. Monkey patching is powerful, but the code that does the actual patching is very tightly coupled with the program to be patched, often handling private and undocumented parts.

Besides being an example of monkey patching, Example 11-6 highlights that protocols are dynamic: `random.shuffle` doesn't care what type of argument it gets, it only needs the object to implement part of the mutable sequence protocol. It doesn't even matter if the object was "born" with the necessary methods or if they were somehow acquired later.

The theme of this chapter so far has been "duck typing": operating with objects regardless of their types, as long as they implement certain protocols.

When we did present diagrams with ABCs, the intent was to show how the protocols are related to the explicit interfaces documented in the abstract classes, but we did not actually inherit from any ABC so far.

In the following sections, we will leverage ABCs directly, and not just as documentation.

Alex Martelli's Waterfowl

After reviewing the usual protocol-style interfaces of Python, we move to ABCs. But before diving into examples and details, Alex Martelli explains in a guest essay why ABCs were a great addition to Python.

I am very grateful to Alex Martelli. He was already the most cited person in this book before he became one of the technical editors. His insights have been invaluable, and then he offered to write this essay. We are incredibly lucky to have him. Take it away, Alex!

Waterfowl and ABCs

By Alex Martelli

I've been credited on Wikipedia for helping spread the helpful meme and sound-bite "*duck typing*" (i.e, ignoring an object's actual type, focusing instead on ensuring that the object implements the method names, signatures, and semantics required for its intended use).

In Python, this mostly boils down to avoiding the use of `isinstance` to check the object's type (not to mention the even worse approach of checking, for example, whether `type(foo) is bar`—which is rightly anathema as it inhibits even the simplest forms of inheritance!).

The overall *duck typing* approach remains quite useful in many contexts—and yet, in many others, an often preferable one has evolved over time. And herein lies a tale…

In recent generations, the taxonomy of genus and species (including but not limited to the family of waterfowl known as Anatidae) has mostly been driven by *phenetics*—an approach focused on similarities of morphology and behavior… chiefly, *observable* traits. The analogy to "duck typing" was strong.

However, parallel evolution can often produce similar traits, both morphological and behavioral ones, among species that are actually unrelated, but just happened to evolve in similar, though separate, ecological niches. Similar "accidental similarities" happen in programming, too—for example, consider the classic OOP example:

```
class Artist:
    def draw(self): ...

class Gunslinger:
    def draw(self): ...

class Lottery:
    def draw(self): ...
```

Clearly, the mere existence of a method called `draw`, callable without arguments, is far from sufficient to assure us that two objects `x` and `y` such that `x.draw()` and `y.draw()` can be called are in any way exchangeable or abstractly equivalent—nothing about the similarity of the semantics resulting from such calls can be inferred. Rather, we need a knowledgeable programmer to somehow positively *assert* that such an equivalence holds at some level!

In biology (and other disciplines) this issue has led to the emergence (and, on many facets, the dominance) of an approach that's an alternative to phenetics, known as *cladistics*—focusing taxonomical choices on characteristics that are inherited from common ancestors, rather than ones that are independently evolved. (Cheap and rapid DNA sequencing can make cladistics highly practical in many more cases, in recent years.)

For example, sheldgeese (once classified as being closer to other geese) and shelducks (once classified as being closer to other ducks) are now grouped together within the subfamily Tadornidae (implying they're closer to each other than to any other Anatidae, as they share a closer common ancestor). Furthermore, DNA analysis has shown, in particular, that the white-winged wood duck is not as close to the Muscovy duck (the latter being a shelduck) as similarity in looks and behavior had long suggested—so the wood duck was reclassified into its own genus, and entirely out of the subfamily!

Does this matter? It depends on the context! For such purposes as deciding how best to cook a waterfowl once you've bagged it, for example, specific observable traits (not all of them—plumage, for example, is de minimis in such a context), mostly texture and flavor (old-fashioned phenetics!), may be far more relevant than cladistics. But for other issues, such as susceptibility to different pathogens (whether you're trying to raise waterfowl in captivity, or preserve them in the wild), DNA closeness can matter much more…

So, by very loose analogy with these taxonomic revolutions in the world of waterfowls, I'm recommending supplementing (not entirely replacing—in certain contexts it shall still serve) good old *duck typing* with… *goose typing*!

What *goose typing* means is: `isinstance(obj, cls)` is now just fine… as long as `cls` is an abstract base class—in other words, `cls`'s metaclass is `abc.ABCMeta`.

You can find many useful existing abstract classes in `collections.abc` (and additional ones in the `numbers` module of *The Python Standard Library*).[3]

Among the many conceptual advantages of ABCs over concrete classes (e.g., Scott Meyer's "all non-leaf classes should be abstract"—see Item 33 in his book, *More Effective C++*), Python's ABCs add one major practical advantage: the `register` class method, which lets end-user code "declare" that a certain class becomes a "virtual" subclass of an ABC (for this purpose the registered class must meet the ABC's method name and signature requirements, and more importantly the underlying semantic contract—but it need not have been developed with any awareness of the ABC, and in particular need not inherit from it!). This goes a long way toward breaking the rigidity and strong coupling that make inheritance something to use with much more caution than typically practiced by most OOP programmers…

Sometimes you don't even need to register a class for an ABC to recognize it as a subclass!

That's the case for the ABCs whose essence boils down to a few special methods. For example:

```
>>> class Struggle:
...     def __len__(self): return 23
...
>>> from collections import abc
>>> isinstance(Struggle(), abc.Sized)
True
```

As you see, `abc.Sized` recognizes `Struggle` as "a subclass," with no need for registration, as implementing the special method named `__len__` is all it takes (it's supposed to be implemented with the proper syntax—callable without arguments—and semantics—returning a nonnegative integer denoting an object's "length"; any code that implements a specially named method, such as `__len__`, with arbitrary, non-compliant syntax and semantics has much worse problems anyway).

So, here's my valediction: whenever you're implementing a class embodying any of the concepts represented in the ABCs in `numbers`, `collections.abc`, or other framework you may be using, be sure (if needed) to subclass it from, or register it into, the corresponding ABC. At the start of your programs using some library or framework defining classes which have omitted to do that, perform the registrations yourself; then, when you must check for (most typically) an argument being, e.g, "a sequence," check whether:

```
isinstance(the_arg, collections.abc.Sequence)
```

3. You can also, of course, define your own ABCs—but I would discourage all but the most advanced Pythonistas from going that route, just as I would discourage them from defining their own custom metaclasses… and even for said "most advanced Pythonistas," those of us sporting deep mastery of every fold and crease in the language, these are not tools for frequent use: such "deep metaprogramming," if ever appropriate, is intended for authors of broad frameworks meant to be independently extended by vast numbers of separate development teams… less than 1% of "most advanced Pythonistas" may ever need that! — *A.M.*

> And, *don't* define custom ABCs (or metaclasses) in production code… if you feel the urge to do so, I'd bet it's likely to be a case of "all problems look like a nail"-syndrome for somebody who just got a shiny new hammer—you (and future maintainers of your code) will be much happier sticking with straightforward and simple code, eschewing such depths. *Valē!*

Besides coining the "goose typing," Alex makes the point that inheriting from an ABC is more than implementing the required methods: it's also a clear declaration of intent by the developer. That intent can also be made explicit through registering a virtual subclass.

In addition, the use of `isinstance` and `issubclass` becomes more acceptable to test against ABCs. In the past, these functions worked against duck typing, but with ABCs they become more flexible. After all, if a component does not implement an ABC by subclassing, it can always be registered after the fact so it passes those explicit type checks.

However, even with ABCs, you should beware that excessive use of `isinstance` checks may be a *code smell*—a symptom of bad OO design. It's usually *not* OK to have a chain of `if/elif/elif` with `insinstance` checks performing different actions depending on the type of an object: you should be using polymorphism for that—i.e., designing your classes so that the interpreter dispatches calls to the proper methods, instead of you hardcoding the dispatch logic in `if/elif/elif` blocks.

There is a common, practical exception to the preceding recommendation: some Python APIs accept a single `str` or a sequence of `str` items; if it's just a single `str`, you want to wrap it in a `list`, to ease processing. Because `str` is a sequence type, the simplest way to distinguish it from any other immutable sequence is to do an explicit `isinstance(x, str)` check.[4]

On the other hand, it's usually OK to perform an `insinstance` check against an ABC if you must enforce an API contract: "Dude, you have to implement this if you want to call me," as technical reviewer Lennart Regebro put it. That's particularly useful in systems that have a plug-in architecture. Outside of frameworks, duck typing is often simpler and more flexible than type checks.

4. Unfortunately, in Python 3.4, there is no ABC that helps distinguish a `str` from `tuple` or other immutable sequences, so we must test against `str`. In Python 2, the `basestr` type exists to help with tests like these. It's not an ABC, but it's a superclass of both `str` and `unicode`; however, in Python 3, `basestr` is gone. Curiously, there is in Python 3 a `collections.abc.ByteString` type, but it only helps detecting `bytes` and `bytearray`.

For example, in several classes in this book, when I needed to take a sequence of items and process them as a `list`, instead of requiring a `list` argument by type checking, I simply took the argument and immediately built a `list` from it: that way I can accept any iterable, and if the argument is not iterable, the call will fail soon enough with a very clear message. One example of this code pattern is in the `__init__` method in Example 11-13, later in this chapter. Of course, this approach wouldn't work if the sequence argument shouldn't be copied, either because it's too large or because my code needs to change it in place. Then an `isinstance(x, abc.MutableSequence)` would be better. If any iterable is acceptable, then calling `iter(x)` to obtain an iterator would be the way to go, as we'll see in "Why Sequences Are Iterable: The iter Function" on page 404.

Another example is how you might imitate the handling of the `field_names` argument in `collections.namedtuple`: `field_names` accepts a single string with identifiers separated by spaces or commas, or a sequence of identifiers. It might be tempting to use `isinstance`, but Example 11-7 shows how I'd do it using duck typing.[5]

Example 11-7. Duck typing to handle a string or an iterable of strings

```
try:  ❶
    field_names = field_names.replace(',', ' ').split()  ❷
except AttributeError:  ❸
    pass  ❹
field_names = tuple(field_names)  ❺
```

❶ Assume it's a string (EAFP = it's easier to ask forgiveness than permission).

❷ Convert commas to spaces and split the result into a list of names.

❸ Sorry, `field_names` doesn't quack like a `str`… there's either no `.replace`, or it returns something we can't `.split`.

❹ Now we assume it's already an iterable of names.

❺ To make sure it's an iterable and to keep our own copy, create a tuple out of what we have.

Finally, in his essay, Alex reinforces more than once the need for restraint in the creation of ABCs. An ABC epidemic would be disastrous, imposing excessive ceremony in a language that became popular because it's practical and pragmatic. During the *Fluent Python* review process, Alex wrote:

> ABCs are meant to encapsulate very general concepts, abstractions, introduced by a framework—things like "a sequence" and "an exact number." [Readers] most likely don't need to write any new ABCs, just use existing ones correctly, to get 99.9% of the benefits without serious risk of misdesign.

5. This snippet was extracted from Example 21-2.

Now let's see goose typing in practice.

Subclassing an ABC

Following Martelli's advice, we'll leverage an existing ABC, `collections.MutableSequence`, before daring to invent our own. In Example 11-8, `FrenchDeck2` is explicitly declared a subclass of `collections.MutableSequence`.

Example 11-8. frenchdeck2.py: FrenchDeck2, a subclass of collections.MutableSequence

```
import collections

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

class FrenchDeck2(collections.MutableSequence):
    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]

    def __setitem__(self, position, value):  # ❶
        self._cards[position] = value

    def __delitem__(self, position):  # ❷
        del self._cards[position]

    def insert(self, position, value):  # ❸
        self._cards.insert(position, value)
```

❶ `__setitem__` is all we need to enable shuffling…

❷ But subclassing `MutableSequence` forces us to implement `__delitem__`, an abstract method of that ABC.

❸ We are also required to implement `insert`, the third abstract method of `MutableSequence`.

Python does not check for the implementation of the abstract methods at import time (when the *frenchdeck2.py* module is loaded and compiled), but only at runtime when we actually try to instantiate `FrenchDeck2`. Then, if we fail to implement any abstract method, we get a `TypeError` exception with a message such as `"Can't instantiate`

`abstract class FrenchDeck2 with abstract methods __delitem__, insert"`. That's why we must implement `__delitem__` and `insert`, even if our `FrenchDeck2` examples do not need those behaviors: the `MutableSequence` ABC demands them.

As Figure 11-2 shows, not all methods of the `Sequence` and `MutableSequence` ABCs are abstract.

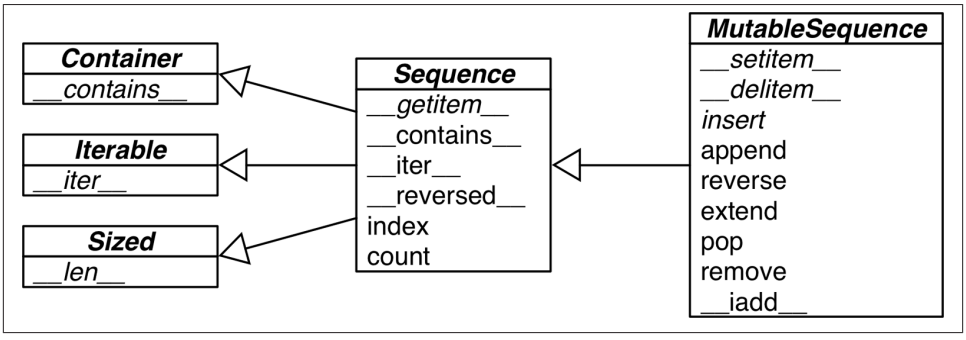

Figure 11-2. UML class diagram for the MutableSequence ABC and its superclasses from collections.abc (inheritance arrows point from subclasses to ancestors; names in italic are abstract classes and abstract methods)

From `Sequence`, `FrenchDeck2` inherits the following ready-to-use concrete methods: `__contains__`, `__iter__`, `__reversed__`, `index`, and `count`. From `MutableSequence`, it gets `append`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`.

The concrete methods in each `collections.abc` ABC are implemented in terms of the public interface of the class, so they work without any knowledge of the internal structure of instances.

As the coder of a concrete subclass, you may be able to override methods inherited from ABCs with more efficient implementations. For example, `__contains__` works by doing a full scan of the sequence, but if your concrete sequence keeps its items sorted, you can write a faster `__contains__` that does a binary search using `bisect` function (see "Managing Ordered Sequences with bisect" on page 44).

To use ABCs well, you need to know what's available. We'll review the collections ABCs next.

ABCs in the Standard Library

Since Python 2.6, ABCs are available in the standard library. Most are defined in the `collections.abc` module, but there are others. You can find ABCs in the `numbers` and `io` packages, for example. But the most widely used is `collections.abc`. Let's see what is available there.

ABCs in collections.abc

There are two modules named `abc` in the standard library. Here we are talking about `collections.abc`. To reduce loading time, in Python 3.4, it's implemented outside of the `collections` package, in *Lib/_collections_abc.py*), so it's imported separately from `collections`. The other `abc` module is just `abc` (i.e., *Lib/abc.py*) where the `abc.ABC` class is defined. Every ABC depends on it, but we don't need to import it ourselves except to create a new ABC.

Figure 11-3 is a summary UML class diagram (without attribute names) of all 16 ABCs defined in `collections.abc` as of Python 3.4. The official documentation of `collections.abc` has a nice table summarizing the ABCs, their relationships, and their abstract and concrete methods (called "mixin methods"). There is plenty of multiple inheritance going on in Figure 11-3. We'll devote most of Chapter 12 to multiple inheritance, but for now it's enough to say that it is usually not a problem when ABCs are concerned.[6]

6. Multiple inheritance was *considered harmful* and excluded from Java, except for interfaces: Java interfaces can extend multiple interfaces, and Java classes can implement multiple interfaces.

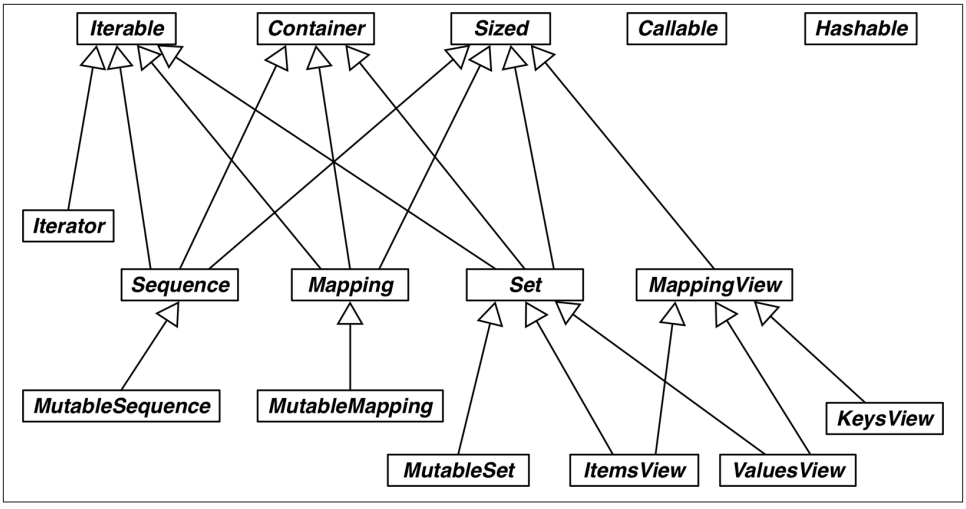

Figure 11-3. UML class diagram for ABCs in collections.abc

Let's review the clusters in Figure 11-3:

`Iterable`, `Container`, *and* `Sized`
: Every collection should either inherit from these ABCs or at least implement compatible protocols. `Iterable` supports iteration with `__iter__`, `Container` supports the `in` operator with `__contains__`, and `Sized` supports `len()` with `__len__`.

`Sequence`, `Mapping`, *and* `Set`
: These are the main immutable collection types, and each has a mutable subclass. A detailed diagram for `MutableSequence` is in Figure 11-2; for `MutableMapping` and `MutableSet`, there are diagrams in Chapter 3 (Figures 3-1 and 3-2).

`MappingView`
: In Python 3, the objects returned from the mapping methods `.items()`, `.keys()`, and `.values()` inherit from `ItemsView`, `ValuesView`, and `ValuesView`, respectively. The first two also inherit the rich interface of `Set`, with all the operators we saw in "Set Operations" on page 82.

`Callable` *and* `Hashable`
: These ABCs are not so closely related to collections, but `collections.abc` was the first package to define ABCs in the standard library, and these two were deemed important enough to be included. I've never seen subclasses of either `Callable` or

`Hashable`. Their main use is to support the `insinstance` built-in as a safe way of determining whether an object is callable or hashable.[7]

`Iterator`
: Note that iterator subclasses `Iterable`. We discuss this further in Chapter 14.

After the `collections.abc` package, the most useful package of ABCs in the standard library is `numbers`, covered next.

The Numbers Tower of ABCs

The `numbers` package defines the so-called "numerical tower" (i.e., this linear hierarchy of ABCs), where `Number` is the topmost superclass, `Complex` is its immediate subclass, and so on, down to `Integral`:

- `Number`
- `Complex`
- `Real`
- `Rational`
- `Integral`

So if you need to check for an integer, use `isinstance(x, numbers.Integral)` to accept `int`, `bool` (which subclasses `int`) or other integer types that may be provided by external libraries that register their types with the `numbers` ABCs. And to satisfy your check, you or the users of your API may always register any compatible type as a virtual subclass of `numbers.Integral`.

If, on the other hand, a value can be a floating-point type, you write `isinstance(x, numbers.Real)`, and your code will happily take `bool`, `int`, `float`, `fractions.Fraction`, or any other noncomplex numerical type provided by an external library, such as NumPy, which is suitably registered.

Somewhat surprisingly, `decimal.Decimal` is not registered as a virtual subclass of `numbers.Real`. The reason is that, if you need the precision of `Decimal` in your program, then you want to be protected from accidental mixing of decimals with other less precise numeric types, particularly floats.

7. For callable detection, there is the `callable()` built-in function—but there is no equivalent `hashable()` function, so `isinstance(my_obj, Hashable)` is the preferred way to test for a hashable object.

After looking at some existing ABCs, let's practice goose typing by implementing an ABC from scratch and putting it to use. The goal here is not to encourage everyone to start coding ABCs left and right, but to learn how to read the source code of the ABCs you'll find in the standard library and other packages.

Defining and Using an ABC

To justify creating an ABC, we need to come up with a context for using it as an extension point in a framework. So here is our context: imagine you need to display advertisements on a website or a mobile app in random order, but without repeating an ad before the full inventory of ads is shown. Now let's assume we are building an ad management framework called `ADAM`. One of its requirements is to support user-provided nonrepeating random-picking classes.[8] To make it clear to `ADAM` users what is expected of a "nonrepeating random-picking" component, we'll define an ABC.

Taking a clue from "stack" and "queue" (which describe abstract interfaces in terms of physical arrangements of objects), I will use a real-world metaphor to name our ABC: bingo cages and lottery blowers are machines designed to pick items at random from a finite set, without repeating, until the set is exhausted.

The ABC will be named `Tombola`, after the Italian name of bingo and the tumbling container that mixes the numbers.[9]

The `Tombola` ABC has four methods. The two abstract methods are:

- `.load(…)`: put items into the container.
- `.pick()`: remove one item at random from the container, returning it.

The concrete methods are:

- `.loaded()`: return `True` if there is at least one item in the container.
- `.inspect()`: return a sorted `tuple` built from the items currently in the container, without changing its contents (its internal ordering is not preserved).

Figure 11-4 shows the `Tombola` ABC and three concrete implementations.

8. Perhaps the client needs to audit the randomizer; or the agency wants to provide a rigged one. You never know…

9. The Oxford English Dictionary defines tombola as "A kind of lottery resembling lotto."

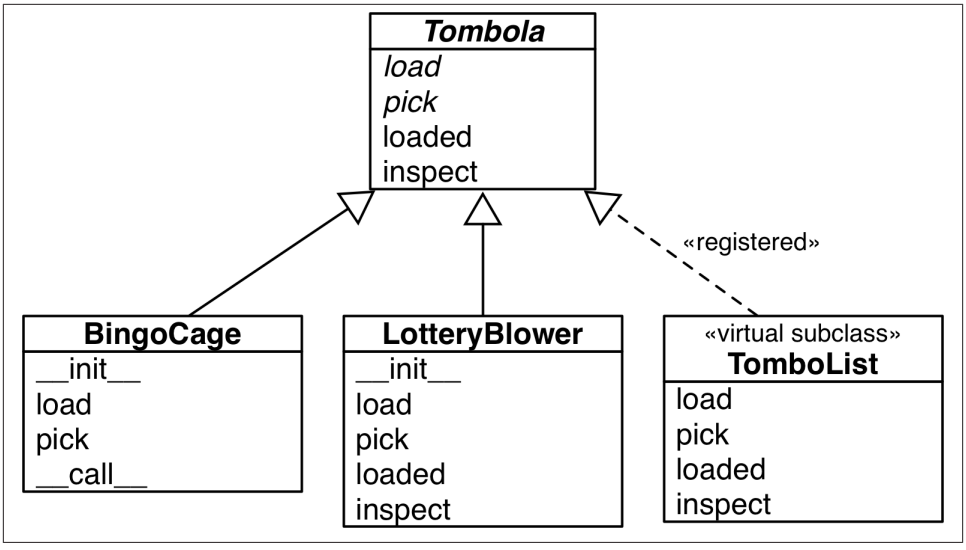

Figure 11-4. UML diagram for an ABC and three subclasses. The name of the Tombola ABC and its abstract methods are written in italics, per UML conventions. The dashed arrow is used for interface implementation, here we are using it to show that TomboList is a virtual subclass of Tombola because it is registered, as we will see later in this chapter.[10]

Example 11-9 shows the definition of the `Tombola` ABC.

Example 11-9. tombola.py: Tombola is an ABC with two abstract methods and two concrete methods

```
import abc

class Tombola(abc.ABC):   ❶

    @abc.abstractmethod
    def load(self, iterable):   ❷
        """Add items from an iterable."""

    @abc.abstractmethod
    def pick(self):   ❸
        """Remove item at random, returning it.

        This method should raise `LookupError` when the instance is empty.
        """
```

10. «registered» and «virtual subclass» are not standard UML words. We are using them to represent a class relationship that is specific to Python.

```
    def loaded(self):   ❹
        """Return `True` if there's at least 1 item, `False` otherwise."""
        return bool(self.inspect())   ❺


    def inspect(self):
        """Return a sorted tuple with the items currently inside."""
        items = []
        while True:   ❻
            try:
                items.append(self.pick())
            except LookupError:
                break
        self.load(items)   ❼
        return tuple(sorted(items))
```

❶ To define an ABC, subclass `abc.ABC`.

❷ An abstract method is marked with the `@abstractmethod` decorator, and often its body is empty except for a docstring.[11]

❸ The docstring instructs implementers to raise `LookupError` if there are no items to pick.

❹ An ABC may include concrete methods.

❺ Concrete methods in an ABC must rely only on the interface defined by the ABC (i.e., other concrete or abstract methods or properties of the ABC).

❻ We can't know how concrete subclasses will store the items, but we can build the `inspect` result by emptying the `Tombola` with successive calls to `.pick()`…

❼ …then use `.load(…)` to put everything back.

An abstract method can actually have an implementation. Even if it does, subclasses will still be forced to override it, but they will be able to invoke the abstract method with `super()`, adding functionality to it instead of implementing from scratch. See the `abc` module documentation for details on `@abstractmethod` usage.

The `.inspect()` method in Example 11-9 is perhaps a silly example, but it shows that, given `.pick()` and `.load(…)` we can inspect what's inside the `Tombola` by picking all items and loading them back. The point of this example is to highlight that it's OK to provide concrete methods in ABCs, as long as they only depend on other methods in the interface. Being aware of their internal data structures, concrete subclasses of `Tom`

11. Before ABCs existed, abstract methods would use the statement `raise NotImplementedError` to signal that subclasses were responsible for their implementation.

`bola` may always override `.inspect()` with a smarter implementation, but they don't have to.

The `.loaded()` method in Example 11-9 may not be as silly, but it's expensive: it calls `.inspect()` to build the sorted `tuple` just to apply `bool()` on it. This works, but a concrete subclass can do much better, as we'll see.

Note that our roundabout implementation of `.inspect()` requires that we catch a `LookupError` thrown by `self.pick()`. The fact that `self.pick()` may raise `LookupError` is also part of its interface, but there is no way to declare this in Python, except in the documentation (see the docstring for the abstract `pick` method in Example 11-9.)

I chose the `LookupError` exception because of its place in the Python hierarchy of exceptions in relation to `IndexError` and `KeyError`, the most likely exceptions to be raised by the data structures used to implement a concrete `Tombola`. Therefore, implementations can raise `LookupError`, `IndexError`, or `KeyError` to comply. See Example 11-10 (for a complete tree, see "5.4. Exception hierarchy" of *The Python Standard Library*).

Example 11-10. Part of the Exception class hierarchy

```
BaseException
 ├── SystemExit
 ├── KeyboardInterrupt
 ├── GeneratorExit
 └── Exception
      ├── StopIteration
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
      ├── EOFError
      ├── ImportError
      ├── LookupError  ❶
      │    ├── IndexError  ❷
      │    └── KeyError  ❸
      ├── MemoryError
      ... etc.
```

❶ `LookupError` is the exception we handle in `Tombola.inspect`.

❷ `IndexError` is the `LookupError` subclass raised when we try to get an item from a sequence with an index beyond the last position.

❸ `KeyError` is raised when we use a nonexistent key to get an item from a mapping.

We now have our very own `Tombola` ABC. To witness the interface checking performed by an ABC, let's try to fool `Tombola` with a defective implementation in Example 11-11.

Example 11-11. A fake Tombola doesn't go undetected

```
>>> from tombola import Tombola
>>> class Fake(Tombola):  # ❶
...     def pick(self):
...         return 13
...
>>> Fake  # ❷
<class '__main__.Fake'>
<class 'abc.ABC'>, <class 'object'>)
>>> f = Fake()  # ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Fake with abstract methods load
```

❶ Declare `Fake` as a subclass of `Tombola`.

❷ The class was created, no errors so far.

❸ `TypeError` is raised when we try to instantiate `Fake`. The message is very clear: `Fake` is considered abstract because it failed to implement `load`, one of the abstract methods declared in the `Tombola` ABC.

So we have our first ABC defined, and we put it to work validating a class. We'll soon subclass the `Tombola` ABC, but first we must cover some ABC coding rules.

ABC Syntax Details

The best way to declare an ABC is to subclass `abc.ABC` or any other ABC.

However, the `abc.ABC` class is new in Python 3.4, so if you are using an earlier version of Python—and it does not make sense to subclass another existing ABC—then you must use the `metaclass=` keyword in the `class` statement, pointing to `abc.ABCMeta` (not `abc.ABC`). In Example 11-9, we would write:

```
class Tombola(metaclass=abc.ABCMeta):
    # ...
```

The `metaclass=` keyword argument was introduced in Python 3. In Python 2, you must use the `__metaclass__` class attribute:

```
class Tombola(object):  # this is Python 2!!!
    __metaclass__ = abc.ABCMeta
    # ...
```

We'll explain metaclasses in Chapter 21. For now, let's accept that a metaclass is a special kind of class, and agree that an ABC is a special kind of class; for example, "regular" classes don't check subclasses, so this is a special behavior of ABCs.

Besides the `@abstractmethod`, the `abc` module defines the `@abstractclassmethod`, `@abstractstaticmethod`, and `@abstractproperty` decorators. However, these last three are deprecated since Python 3.3, when it became possible to stack decorators on top of `@abstractmethod`, making the others redundant. For example, the preferred way to declare an abstract class method is:

```
class MyABC(abc.ABC):
    @classmethod
    @abc.abstractmethod
    def an_abstract_classmethod(cls, ...):
        pass
```

The order of stacked function decorators usually matters, and in the case of `@abstractmethod`, the documentation is explicit:

> When abstractmethod() is applied in combination with other method descriptors, it should be applied as the innermost decorator, …[12]

In other words, no other decorator may appear between `@abstractmethod` and the `def` statement.

Now that we got these ABC syntax issues covered, let's put `Tombola` to use by implementing some full-fledged concrete descendants of it.

Subclassing the Tombola ABC

Given the `Tombola` ABC, we'll now develop two concrete subclasses that satisfy its interface. These classes were pictured in Figure 11-4, along with the virtual subclass to be discussed in the next section.

The `BingoCage` class in Example 11-12 is a variation of Example 5-8 using a better randomizer. This `BingoCage` implements the required abstract methods `load` and `pick`, inherits `loaded` from `Tombola`, overrides `inspect`, and adds `__call__`.

Example 11-12. bingo.py: BingoCage is a concrete subclass of Tombola

```
import random

from tombola import Tombola
```

12. `@abc.abstractmethod` entry in the `abc` module documentation.

```
class BingoCage(Tombola):    ❶

    def __init__(self, items):
        self._randomizer = random.SystemRandom()    ❷
        self._items = []
        self.load(items)    ❸

    def load(self, items):
        self._items.extend(items)
        self._randomizer.shuffle(self._items)    ❹

    def pick(self):    ❺
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage')

    def __call__(self):    ❻
        self.pick()
```

❶ This `BingoCage` class explicitly extends `Tombola`.

❷ Pretend we'll use this for online gaming. `random.SystemRandom` implements the `random` API on top of the `os.urandom(…)` function, which provides random bytes "suitable for cryptographic use" according to the `os` module docs.

❸ Delegate initial loading to the `.load(…)` method.

❹ Instead of the plain `random.shuffle()` function, we use the `.shuffle()` method of our `SystemRandom` instance.

❺ `pick` is implemented as in Example 5-8.

❻ `__call__` is also from Example 5-8. It's not needed to satisfy the `Tombola` interface, but there's no harm in adding extra methods.

`BingoCage` inherits the expensive `loaded` and the silly `inspect` methods from `Tombola`. Both could be overridden with much faster one-liners, as in Example 11-13. The point is: we can be lazy and just inherit the suboptimal concrete methods from an ABC. The methods inherited from `Tombola` are not as fast as they could be for `BingoCage`, but they do provide correct results for any `Tombola` subclass that correctly implements `pick` and `load`.

Example 11-13 shows a very different but equally valid implementation of the `Tombola` interface. Instead of shuffling the "balls" and popping the last, `LotteryBlower` pops from a random position.

Example 11-13. lotto.py: LotteryBlower is a concrete subclass that overrides the inspect and loaded methods from Tombola

```
import random

from tombola import Tombola


class LotteryBlower(Tombola):

    def __init__(self, iterable):
        self._balls = list(iterable)   ❶

    def load(self, iterable):
        self._balls.extend(iterable)

    def pick(self):
        try:
            position = random.randrange(len(self._balls))   ❷
        except ValueError:
            raise LookupError('pick from empty BingoCage')
        return self._balls.pop(position)   ❸

    def loaded(self):   ❹
        return bool(self._balls)

    def inspect(self):   ❺
        return tuple(sorted(self._balls))
```

❶ The initializer accepts any iterable: the argument is used to build a list.

❷ The `random.randrange(…)` function raises `ValueError` if the range is empty, so we catch that and throw `LookupError` instead, to be compatible with `Tombola`.

❸ Otherwise the randomly selected item is popped from `self._balls`.

❹ Override `loaded` to avoid calling `inspect` (as `Tombola.loaded` does in Example 11-9). We can make it faster by working with `self._balls` directly—no need to build a whole sorted `tuple`.

❺ Override `inspect` with one-liner.

Example 11-13 illustrates an idiom worth mentioning: in `__init__`, `self._balls` stores `list(iterable)` and not just a reference to `iterable` (i.e., we did not merely assign `iterable` to `self._balls`). As mentioned before,[13] this makes our `LotteryBlower` flexible because the `iterable` argument may be any iterable type. At the same time, we make sure to store its items in a `list` so we can `pop` items. And even if we always get lists as the `iterable` argument, `list(iterable)` produces a copy of the argument,

13. I gave this as an example of duck typing after Martelli's "Waterfowl and ABCs" on page 314.

which is a good practice considering we will be removing items from it and the client may not be expecting the `list` of items she provided to be changed.[14]

We now come to the crucial dynamic feature of goose typing: declaring virtual subclasses with the `register` method.

A Virtual Subclass of Tombola

An essential characteristic of goose typing—and the reason why it deserves a waterfowl name—is the ability to register a class as a *virtual subclass* of an ABC, even if it does not inherit from it. When doing so, we promise that the class faithfully implements the interface defined in the ABC—and Python will believe us without checking. If we lie, we'll be caught by the usual runtime exceptions.

This is done by calling a `register` method on the ABC. The registered class then becomes a virtual subclass of the ABC, and will be recognized as such by functions like `issubclass` and `isinstance`, but it will not inherit any methods or attributes from the ABC.

Virtual subclasses do not inherit from their registered ABCs, and are not checked for conformance to the ABC interface at any time, not even when they are instantiated. It's up to the subclass to actually implement all the methods needed to avoid runtime errors.

The `register` method is usually invoked as a plain function (see "Usage of register in Practice" on page 338), but it can also be used as a decorator. In Example 11-14, we use the decorator syntax and implement `TomboList`, a virtual subclass of `Tombola` depicted in Figure 11-5.

`TomboList` works as advertised, and the doctests that prove it are described in "How the Tombola Subclasses Were Tested" on page 335.

14. "Defensive Programming with Mutable Parameters" on page 232 in Chapter 8 was devoted to the aliasing issue we just avoided here.

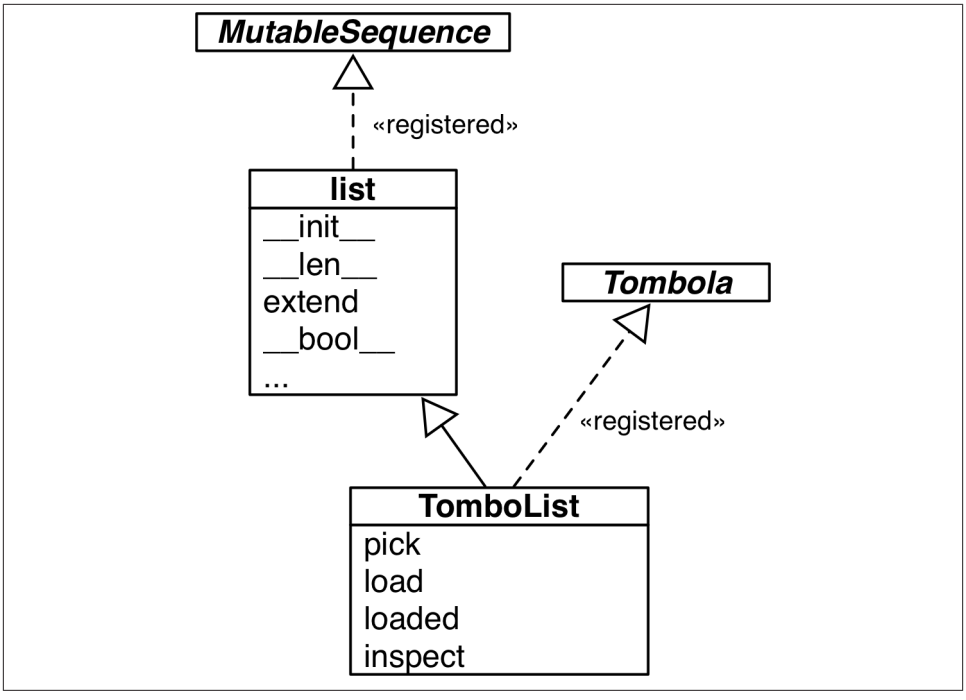

Figure 11-5. UML class diagram for the TomboList, a real subclass of list and a virtual subclass of Tombola

Example 11-14. tombolist.py: class TomboList is a virtual subclass of Tombola

```
from random import randrange

from tombola import Tombola

@Tombola.register  # ❶
class TomboList(list):  # ❷

    def pick(self):
        if self:  # ❸
            position = randrange(len(self))
            return self.pop(position)  # ❹
        else:
            raise LookupError('pop from empty TomboList')

    load = list.extend  # ❺

    def loaded(self):
        return bool(self)  # ❻

    def inspect(self):
        return tuple(sorted(self))
```

```
# Tombola.register(TomboList)  # ❼
```

❶ `Tombolist` is registered as a virtual subclass of `Tombola`.

❷ `Tombolist` extends `list`.

❸ `Tombolist` inherits `__bool__` from `list`, and that returns `True` if the list is not empty.

❹ Our `pick` calls `self.pop`, inherited from `list`, passing a random item index.

❺ `Tombolist.load` is the same as `list.extend`.

❻ `loaded` delegates to `bool`.[15]

❼ If you're using Python 3.3 or earlier, you can't use `.register` as a class decorator. You must use standard call syntax.

Note that because of the registration, the functions `issubclass` and `isinstance` act as if `TomboList` is a subclass of `Tombola`:

```
>>> from tombola import Tombola
>>> from tombolist import TomboList
>>> issubclass(TomboList, Tombola)
True
>>> t = TomboList(range(100))
>>> isinstance(t, Tombola)
True
```

However, inheritance is guided by a special class attribute named `__mro__`—the Method Resolution Order. It basically lists the class and its superclasses in the order Python uses to search for methods.[16] If you inspect the `__mro__` of `TomboList`, you'll see that it lists only the "real" superclasses—`list` and `object`:

```
>>> TomboList.__mro__
(<class 'tombolist.TomboList'>, <class 'list'>, <class 'object'>)
```

`Tombola` is not in `Tombolist.__mro__`, so `Tombolist` does not inherit any methods from `Tombola`.

15. The same trick I used with `load` doesn't work with `loaded`, because the `list` type does not implement `__bool__`, the method I'd have to bind to `loaded`. On the other hand, the `bool` built-in function doesn't need `__bool__` to work because it can also use `__len__`. See "4.1. Truth Value Testing" in the "Built-in Types" chapter.
16. There is a whole section explaining the `__mro__` class attribute in "Multiple Inheritance and Method Resolution Order" on page 351. Right now, this quick explanation will do.

As I coded different classes to implement the same interface, I wanted a way to submit them all to the same suite of doctests. The next section shows how I leveraged the API of regular classes and ABCs to do it.

How the Tombola Subclasses Were Tested

The script I used to test the `Tombola` examples uses two class attributes that allow introspection of a class hierarchy:

`__subclasses__()`
: Method that returns a list of the immediate subclasses of the class. The list does not include virtual subclasses.

`_abc_registry`
: Data attribute—available only in ABCs—that is bound to a `WeakSet` with weak references to registered virtual subclasses of the abstract class.

To test all `Tombola` subclasses, I wrote a script to iterate over a list built from `Tombola.__subclasses__()` and `Tombola._abc_registry`, and bind each class to the name `ConcreteTombola` used in the doctests.

A successful run of the test script looks like this:

```
$ python3 tombola_runner.py
BingoCage        23 tests,  0 failed - OK
LotteryBlower    23 tests,  0 failed - OK
TumblingDrum     23 tests,  0 failed - OK
TomboList        23 tests,  0 failed - OK
```

The test script is Example 11-15 and the doctests are in Example 11-16.

Example 11-15. tombola_runner.py: test runner for Tombola subclasses

```
import doctest

from tombola import Tombola

# modules to test
import bingo, lotto, tombolist, drum   ❶

TEST_FILE = 'tombola_tests.rst'
TEST_MSG = '{0:16} {1.attempted:2} tests, {1.failed:2} failed - {2}'


def main(argv):
    verbose = '-v' in argv
    real_subclasses = Tombola.__subclasses__()   ❷
    virtual_subclasses = list(Tombola._abc_registry)   ❸

    for cls in real_subclasses + virtual_subclasses:   ❹
        test(cls, verbose)
```

```
def test(cls, verbose=False):

    res = doctest.testfile(
            TEST_FILE,
            globs={'ConcreteTombola': cls},   ❺
            verbose=verbose,
            optionflags=doctest.REPORT_ONLY_FIRST_FAILURE)
    tag = 'FAIL' if res.failed else 'OK'
    print(TEST_MSG.format(cls.__name__, res, tag))   ❻


if __name__ == '__main__':
    import sys
    main(sys.argv)
```

❶ Import modules containing real or virtual subclasses of `Tombola` for testing.

❷ `__subclasses__()` lists the direct descendants that are alive in memory. That's why we imported the modules to test, even if there is no further mention of them in the source code: to load the classes into memory.

❸ Build a `list` from `_abc_registry` (which is a `WeakSet`) so we can concatenate it with the result of `__subclasses__()`.

❹ Iterate over the subclasses found, passing each to the `test` function.

❺ The `cls` argument—the class to be tested—is bound to the name `ConcreteTombola` in the global namespace provided to run the doctest.

❻ The test result is printed with the name of the class, the number of tests attempted, tests failed, and an `'OK'` or `'FAIL'` label.

The doctest file is Example 11-16.

Example 11-16. tombola_tests.rst: doctests for Tombola subclasses

```
==============
Tombola tests
==============

Every concrete subclass of Tombola should pass these tests.


Create and load instance from iterable::

    >>> balls = list(range(3))
    >>> globe = ConcreteTombola(balls)
    >>> globe.loaded()
    True
    >>> globe.inspect()
    (0, 1, 2)
```

```
Pick and collect balls::

    >>> picks = []
    >>> picks.append(globe.pick())
    >>> picks.append(globe.pick())
    >>> picks.append(globe.pick())


Check state and results::

    >>> globe.loaded()
    False
    >>> sorted(picks) == balls
    True


Reload::

    >>> globe.load(balls)
    >>> globe.loaded()
    True
    >>> picks = [globe.pick() for i in balls]
    >>> globe.loaded()
    False


Check that `LookupError` (or a subclass) is the exception
thrown when the device is empty::

    >>> globe = ConcreteTombola([])
    >>> try:
    ...     globe.pick()
    ... except LookupError as exc:
    ...     print('OK')
    OK


Load and pick 100 balls to verify that they all come out::

    >>> balls = list(range(100))
    >>> globe = ConcreteTombola(balls)
    >>> picks = []
    >>> while globe.inspect():
    ...     picks.append(globe.pick())
    >>> len(picks) == len(balls)
    True
    >>> set(picks) == set(balls)
    True
```

```
Check that the order has changed and is not simply reversed::

    >>> picks != balls
    True
    >>> picks[::-1] != balls
    True

Note: the previous 2 tests have a *very* small chance of failing
even if the implementation is OK. The probability of the 100
balls coming out, by chance, in the order they were inspect is
1/100!, or approximately 1.07e-158. It's much easier to win the
Lotto or to become a billionaire working as a programmer.

THE END
```

This concludes our `Tombola` ABC case study. In the next section, we'll address how the `register` ABC function is used in the wild.

Usage of register in Practice

In Example 11-14, we used `Tombola.register` as a class decorator. Prior to Python 3.3, `register` could not be used like that—it had to be called as a plain function after the class definition, as suggested by the comment at the end of Example 11-14.

However, even if `register` can now be used as a decorator, it's more widely deployed as a function to register classes defined elsewhere. For example, in the source code for the `collections.abc` module, the built-in types `tuple`, `str`, `range`, and `memoryview` are registered as virtual subclasses of `Sequence` like this:

```
Sequence.register(tuple)
Sequence.register(str)
Sequence.register(range)
Sequence.register(memoryview)
```

Several other built-in types are registered to ABCs in *_collections_abc.py*. Those registrations happen only when that module is imported, which is OK because you'll have to import it anyway to get the ABCs: you need access to `MutableMapping` to be able to write `isinstance(my_dict, MutableMapping)`.

We'll wrap up this chapter by explaining a bit of ABC magic that Alex Martelli performed in "Waterfowl and ABCs" on page 314.

Geese Can Behave as Ducks

In his *Waterfowl and ABCs* essay, Alex shows that a class can be recognized as a virtual subclass of an ABC even without registration. Here is his example again, with an added test using `issubclass`:

```
>>> class Struggle:
...     def __len__(self): return 23
...
>>> from collections import abc
>>> isinstance(Struggle(), abc.Sized)
True
>>> issubclass(Struggle, abc.Sized)
True
```

Class `Struggle` is considered a subclass of `abc.Sized` by the `issubclass` function (and, consequently, by `isinstance` as well) because `abc.Sized` implements a special class method named `__subclasshook__`. See Example 11-17.

Example 11-17. Sized definition from the source code of Lib/_collections_abc.py (Python 3.4)

```
class Sized(metaclass=ABCMeta):

    __slots__ = ()

    @abstractmethod
    def __len__(self):
        return 0

    @classmethod
    def __subclasshook__(cls, C):
        if cls is Sized:
            if any("__len__" in B.__dict__ for B in C.__mro__):  # ❶
                return True  # ❷
        return NotImplemented  # ❸
```

❶ If there is an attribute named `__len__` in the `__dict__` of any class listed in `C.__mro__` (i.e., `C` and its superclasses)…

❷ …return `True`, signaling that `C` is a virtual subclass of `Sized`.

❸ Otherwise return `NotImplemented` to let the subclass check proceed.

If you are interested in the details of the subclass check, see the source code for the `ABCMeta.__subclasscheck__` method in *Lib/abc.py*. Beware: it has lots of ifs and two recursive calls.

The `__subclasshook__` adds some duck typing DNA to the whole goose typing proposition. You can have formal interface definitions with ABCs, you can make `isinstance` checks everywhere, and still have a completely unrelated class play along just because it implements a certain method (or because it does whatever it takes to convince a `__subclasshook__` to vouch for it). Of course, this only works for ABCs that do provide a `__subclasshook__`.

Is it a good idea to implement `__subclasshook__` in our own ABCs? Probably not. All the implementations of `__subclasshook__` I've seen in the Python source code are in ABCs like `Sized` that declare just one special method, and they simply check for that special method name. Given their "special" status, you can be pretty sure that any method named `__len__` does what you expect. But even in the realm of special methods and fundamental ABCs, it can be risky to make such assumptions. For example, mappings implement `__len__`, `__getitem__`, and `__iter__` but they are rightly not considered a subtype of `Sequence`, because you can't retrieve items using an integer offset and they make no guarantees about the ordering of items—except of course for `OrderedDict`, which preserves the insertion order, but does support item retrieval by offset either.

For ABCs that you and I may write, a `__subclasshook__` would be even less dependable. I am not ready to believe that any class named `Spam` that implements or inherits `load`, `pick`, `inspect`, and `loaded` is guaranteed to behave as a `Tombola`. It's better to let the programmer affirm it by subclassing `Spam` from `Tombola`, or at least registering: `Tombola.register(Spam)`. Of course, your `__subclasshook__` could also check method signatures and other features, but I just don't think it's worthwhile.

Chapter Summary

The goal of this chapter was to travel from the highly dynamic nature of informal interfaces—called protocols—visit the static interface declarations of ABCs, and conclude with the dynamic side of ABCs: virtual subclasses and dynamic subclass detection with `__subclasshook__`.

We started the journey by reviewing the traditional understanding of interfaces in the Python community. For most of the history of Python, we've been mindful of interfaces, but they were informal like the protocols from Smalltalk, and the official docs used language such as "foo protocol," "foo interface," and "foo-like object" interchangeably. Protocol-style interfaces have nothing to do with inheritance; each class stands alone when implementing a protocol. That's what interfaces look like when you embrace duck typing.

With Example 11-3, we observed how deeply Python supports the sequence protocol. If a class implements `__getitem__` and nothing else, Python manages to iterate over it, and the `in` operator just works. We then went back to the old `FrenchDeck` example of Chapter 1 to support shuffling by dynamically adding a method. This illustrated monkey patching and emphasized the dynamic nature of protocols. Again we saw how a partially implemented protocol can be useful: just adding `__setitem__` from the mutable sequence protocol allowed us to leverage a ready-to-use function from the standard library: `random.shuffle`. Being aware of existing protocols lets us make the most of the rich Python standard library.

Alex Martelli then introduced the term "goose typing"[17] to describe a new style of Python programming. With "goose typing," ABCs are used to make interfaces explicit and classes may claim to implement an interface by subclassing an ABC or by registering with it—without requiring the strong and static link of an inheritance relationship.

The `FrenchDeck2` example made clear the main drawbacks and advantages of explicit ABCs. Inheriting from `abc.MutableSequence` forced us to implement two methods we did not really need: `insert` and `__delitem__`. On the other hand, even a Python newbie can look at `FrenchDeck2` and see that it's a mutable sequence. And, as bonus, we inherited 11 ready-to-use methods from `abc.MutableSequence` (five indirectly from `abc.Sequence`).

After a panoramic view of existing ABCs from `collections.abc` in Figure 11-3, we wrote an ABC from scratch. Doug Hellmann, creator of the cool PyMOTW.com (Python Module of the Week) explains the motivation:

> By defining an abstract base class, a common API can be established for a set of subclasses. This capability is especially useful in situations where someone less familiar with the source for an application is going to provide plug-in extensions…[18]

Putting the `Tombola` ABC to work, we created three concrete subclasses: two inheriting from `Tombola`, the other a virtual subclass registered with it, all passing the same suite of tests.

In concluding the chapter, we mentioned how several built-in types are registered to ABCs in the `collections.abc` module so you can ask `isinstance(memoryview, abc.Sequence)` and get `True`, even if `memoryview` does not inherit from `abc.Sequence`. And finally we went over the `__subclasshook__` magic, which lets an ABC recognize any unregistered class as a subclass, as long as it passes a test that can be as simple or as complex as you like—the examples in the standard library merely check for method names.

To sum up, I'd like to restate Alex Martelli's admonition that we should refrain from creating our own ABCs, except when we are building user-extensible frameworks—which most of the time we are not. On a daily basis, our contact with ABCs should be subclassing or registering classes with existing ABCs. Less often than subclassing or registering, we might use ABCs for `isinstance` checks. And even more rarely—if ever—we find occasion to write a new ABC from scratch.

After 15 years of Python, the first abstract class I ever wrote that is not a didactic example was the `Board` class of the Pingo project. The drivers that support different single board computers and controllers are subclasses of `Board`, thus sharing the same interface. In

17. Alex coined the expression "goose typing" and this is the first time ever it appears in a book!

18. PyMOTW, `abc` module page, section "Why use Abstract Base Classes?"

reality, although conceived and implemented as an abstract class, the `pingo.Board` class does not subclass `abc.ABC` as I write this.[19] I intend to make `Board` an explicit ABC eventually—but there are more important things to do in the project.

Here is a fitting quote to end this chapter:

> Although ABCs facilitate type checking, it's not something that you should overuse in a program. At its heart, Python is a dynamic language that gives you great flexibility. Trying to enforce type constraints everywhere tends to result in code that is more complicated than it needs to be. You should embrace Python's flexibility.[20]
>
> — David Beazley and Brian Jones
> *Python Cookbook*

Or, as technical reviewer Leonardo Rochael wrote: "If you feel tempted to create a custom ABC, please first try to solve your problem through regular duck-typing."

Further Reading

Beazley and Jones's *Python Cookbook, 3rd Edition* (O'Reilly) has a section about defining an ABC (Recipe 8.12). The book was written before Python 3.4, so they don't use the now preferred syntax when declaring ABCs by subclassing from `abc.ABC` instead of using the `metaclass` keyword. Apart from this small detail, the recipe covers the major ABC features very well, and ends with the valuable advice quoted at the end of the previous section.

The Python Standard Library by Example by Doug Hellmann (Addison-Wesley), has a chapter about the `abc` module. It's also available on the Web in Doug's excellent PyMOTW — Python Module of the Week. Both the book and the site focus on Python 2; therefore, adjustments must be made if you are using Python 3. And for Python 3.4, remember that the only recommended ABC method decorator is `@abstractmethod`—the others were deprecated. The other quote about ABCs in the chapter summary is from Doug's site and book.

When using ABCs, multiple inheritance is not only common but practically inevitable, because each of the fundamental collection ABCs—`Sequence`, `Mapping`, and `Set`—extends multiple ABCs (see Figure 11-3). Therefore, Chapter 12 is an important follow-up to this one.

PEP 3119 — Introducing Abstract Base Classes gives the rationale for ABCs, and PEP 3141 - A Type Hierarchy for Numbers presents the ABCs of the `numbers` module.

19. You'll find that in the Python standard library too: classes that are in fact abstract but nobody ever made them explicitly so.

20. *Python Cookbook, 3rd Edition* (O'Reilly), "Recipe 8.12. Defining an Interface or Abstract Base Class", p. 276.

For a discussion of the pros and cons of dynamic typing, see Guido van Rossum's interview to Bill Venners in "Contracts in Python: A Conversation with Guido van Rossum, Part IV".

The `zope.interface` package provides a way of declaring interfaces, checking whether objects implement them, registering providers, and querying for providers of a given interface. The package started as a core piece of Zope 3, but it can and has been used outside of Zope. It is the basis of the flexible component architecture of large-scale Python projects like Twisted, Pyramid, and Plone. Lennart Regebro has a great introduction to `zope.interface` in "A Python Component Architecture". Baiju M wrote an entire book about it: *A Comprehensive Guide to Zope Component Architecture*.

Soapbox

Type Hints

Probably the biggest news in the Python world in 2014 was that Guido van Rossum gave a green light to the implementation of optional static type checking using function annotations, similar to what the Mypy checker does. This happened in the Python-ideas mailing-list on August 15. The message is Optional static typing — the crossroads. The next month, PEP 484 - Type Hints was published as a draft, authored by Guido.

The idea is to let programmers optionally use annotations to declare parameter and return types in function definitions. The key word here is *optionally*. You'd only add such annotations if you want the benefits and constraints that come with them, and you could put them in some functions but not in others.

On the surface, this may sound like what Microsoft did with with TypeScript, its JavaScript superset, except that TypeScript goes much further: it adds new language constructs (e.g., modules, classes, explicit interfaces, etc.), allows typed variable declarations, and actually compiles down to plain JavaScript. As of this writing, the goals of optional static typing in Python are much less ambitious.

To understand the reach of this proposal, there is a key point that Guido makes in the historic August 15, 2014, email:

> I am going to make one additional assumption: the main use cases will be linting, IDEs, and doc generation. These all have one thing in common: it should be possible to run a program even though it fails to type check. Also, adding types to a program should not hinder its performance (nor will it help :-).

So, it seems this is not such a radical move as it seems at first. PEP 482 - Literature Overview for Type Hints is referenced by PEP 484 - Type Hints, and briefly documents type hints in third-party Python tools and in other languages.

Radical or not, type hints are upon us: support for PEP 484 in the form of a `typing` module is likely to land in Python 3.5 already. The way the proposal is worded and

implemented makes it clear that no existing code will stop running because of the lack of type hints—or their addition, for that matter.

Finally, PEP 484 clearly states:

> It should also be emphasized that Python will remain a dynamically typed language, and the authors have no desire to ever make type hints mandatory, even by convention.

Is Python Weakly Typed?

Discussions about language typing disciplines are sometimes confused due to lack of a uniform terminology. Some writers (like Bill Venners in the interview with Guido mentioned in "Further Reading" on page 342), say that Python has weak typing, which puts it into the same category of JavaScript and PHP. A better way of talking about typing discipline is to consider two different axes:

Strong versus weak typing
: If the language rarely performs implicit conversion of types, it's considered strongly typed; if it often does it, it's weakly typed. Java, C++, and Python are strongly typed. PHP, JavaScript, and Perl are weakly typed.

Static versus dynamic typing
: If type-checking is performed at compile time, the language is statically typed; if it happens at runtime, it's dynamically typed. Static typing requires type declarations (some modern languages use type inference to avoid some of that). Fortran and Lisp are the two oldest programming languages still alive and they use, respectively, static and dynamic typing.

Strong typing helps catch bugs early.

Here are some examples of why weak typing is bad:[21]

```
// this is JavaScript (tested with Node.js v0.10.33)
'' == '0'   // false
0 == ''     // true
0 == '0'    // true
'' < 0      // false
'' < '0'    // true
```

Python does not perform automatic coercion between strings and numbers, so the `==` expressions all result `False`—preserving the transitivity of `==`—and the `<` comparisons raise `TypeError` in Python 3.

Static typing makes it easier for tools (compilers, IDEs) to analyze code to detect errors and provide other services (optimization, refactoring, etc.). Dynamic typing increases opportunities for reuse, reducing line count, and allows interfaces to emerge naturally as protocols, instead of being imposed early on.

21. Adapted from Douglas Crockford's *JavaScript: The Good Parts* (O'Reilly), Appendix B, p. 109.

To summarize, Python uses dynamic and strong typing. PEP 484 - Type Hints will not change that, but will allow API authors to add optional type annotations so that tools can perform some static type checking.

Monkey Patching

Monkey patching has a bad reputation. If abused, it can lead to systems that are hard to understand and maintain. The patch is usually tightly coupled with its target, making it brittle. Another problem is that two libraries that apply monkey-patches may step on each other's toes, with the second library to run destroying patches of the first.

But monkey patching can also be useful, for example, to make a class implement a protocol at runtime. The adapter design pattern solves the same problem by implementing a whole new class.

It's easy to monkey-patch Python code, but there are limitations. Unlike Ruby and JavaScript, Python does not let you monkey-patch the built-in types. I actually consider this an advantage, because you can be certain that a `str` object will always have those same methods. This limitation reduces the chance that external libraries try to apply conflicting patches.

Interfaces in Java, Go, and Ruby

Since C++ 2.0 (1989), abstract classes have been used to specify interfaces in that language. The designers of Java opted not to have multiple inheritance of classes, which precluded the use of abstract classes as interface specifications—because often a class needs to implement more than one interface. But they added the `interface` as a language construct, and a class can implement more than one interface—a form of multiple inheritance. Making interface definitions more explicit than ever was a great contribution of Java. With Java 8, an interface can provide method implementations, called Default Methods. With this, Java interfaces became closer to abstract classes in C++ and Python.

The Go language has a completely different approach. First of all, there is no inheritance in Go. You can define interfaces, but you don't need (and you actually can't) explicitly say that a certain type implements an interface. The compiler determines that automatically. So what they have in Go could be called "static duck typing," in the sense that interfaces are checked at compile time but what matters is what types actually implement.

Compared to Python, it's as if, in Go, every ABC implemented the `__subclasshook__` checking function names and signatures, and you never subclassed or registered an ABC. If we wanted Python to look more like Go, we would have to perform type checks on all function arguments. Some of the infrastructure is available (recall "Function Annotations" on page 154). Guido has already said he thinks it's OK to use those annotations for type checking—at least in support tools. See "Soapbox" on page 163 in Chapter 5 for more about this.

Rubyists are firm believers in duck typing, and Ruby has no formal way to declare an interface or an abstract class, except to do the same we did in Python prior to 2.6: raise

`NotImplementedError` in the body of methods to make them abstract by forcing the user to subclass and implement them.

Meanwhile, I read that Yukihiro "Matz" Matsumoto, creator of Ruby, said in a keynote in September 2014 that static typing may be in the future of the language. That was at Ruby Kaigi in Japan, one of the most important Ruby conferences every year. As I write this, I haven't seen a transcript, but Godfrey Chan posted about it on his blog: "Ruby Kaigi 2014: Day 2". From Chan's report, it seems Matz focused on function annotations. There is even mention of Python function annotations.

I wonder if function annotations would be really good without ABCs to add structure to the type system without losing flexibility. So maybe formal interfaces are also in the future of Ruby.

I believe Python ABCs, with the `register` function and `__subclasshook__`, brought formal interfaces to the language without throwing away the advantages of dynamic typing.

Perhaps the geese are poised to overtake the ducks.

Metaphors and Idioms in Interfaces

A metaphor fosters understanding by making constraints clear. That's the value of the words "stack" and "queue" in describing those fundamental data structures: they make clear how items can be added or removed. On the other hand, Alan Cooper writes in *About Face, 4E* (Wiley):

> Strict adherence to metaphors ties interfaces unnecessarily tightly to the workings of the physical world.

He's referring to user interfaces, but the admonition applies to APIs as well. But Cooper does grant that when a "truly appropriate" metaphor "falls on our lap," we can use it (he writes "falls on our lap" because it's so hard to find fitting metaphors that you should not spend time actively looking for them). I believe the bingo machine imagery I used in this chapter is appropriate and I stand by it.

About Face is by far the best book about UI design I've read—and I've read a few. Letting go of metaphors as a design paradigm, and replacing it with "idiomatic interfaces" was the most valuable thing I learned from Cooper's work. As mentioned, Cooper does not deal with APIs, but the more I think about his ideas, the more I see how they apply to Python. The fundamental protocols of the language are what Cooper calls "idioms." Once we learn what a "sequence" is we can apply that knowledge in different contexts. This is a main theme of *Fluent Python*: highlighting the fundamental idioms of the language, so your code is concise, effective, and readable—for a fluent Pythonista.