CHAPTER 13

Interfaces, Protocols, and ABCs

Program to an interface, not an implementation.

—Gamma, Helm, Johnson, Vlissides, First Principle of Object-Oriented Design[1]

Object-oriented programming is all about interfaces. The best approach to understanding a type in Python is knowing the methods it provides—its interface—as discussed in "Types Are Defined by Supported Operations" on page 260 (Chapter 8).

Depending on the programming language, we have one or more ways of defining and using interfaces. Since Python 3.8, we have four ways. They are depicted in the *Typing Map* (Figure 13-1). We can summarize them like this:

Duck typing
: Python's default approach to typing from the beginning. We've been studying duck typing since Chapter 1.

Goose typing
: The approach supported by abstract base classes (ABCs) since Python 2.6, which relies on runtime checks of objects against ABCs. *Goose typing* is a major subject in this chapter.

Static typing
: The traditional approach of statically-typed languages like C and Java; supported since Python 3.5 by the `typing` module, and enforced by external type checkers compliant with PEP 484—Type Hints. This is not the theme of this chapter. Most of Chapter 8 and the upcoming Chapter 15 are about static typing.

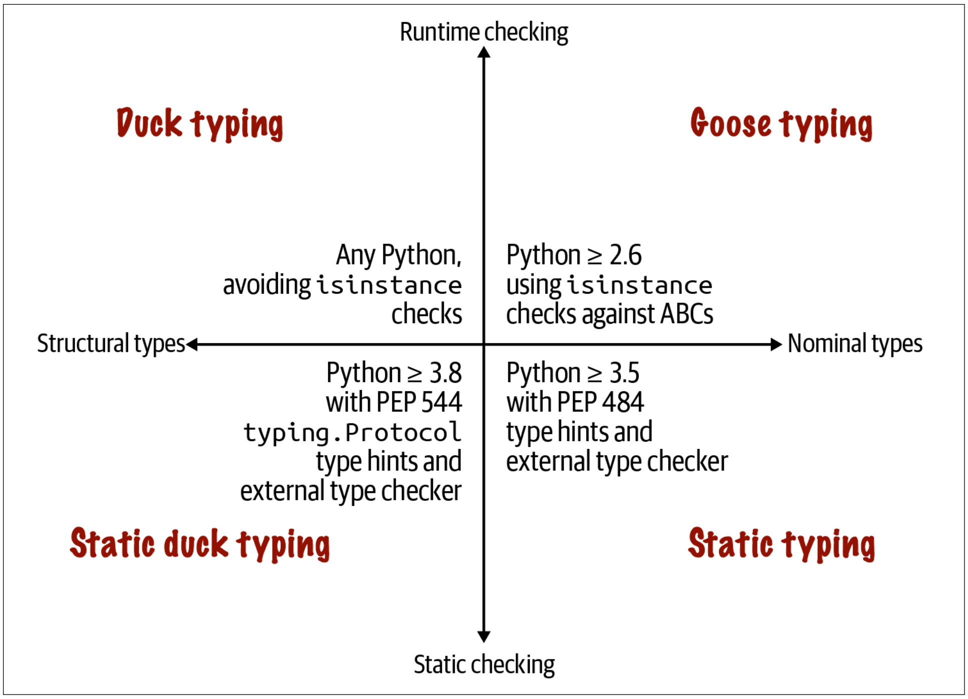
1 *Design Patterns: Elements of Reusable Object-Oriented Software*, "Introduction," p. 18.

Static duck typing

An approach made popular by the Go language; supported by subclasses of `typing.Protocol`—new in Python 3.8—also enforced by external type checkers. We first saw this in "Static Protocols" on page 286 (Chapter 8).

The Typing Map

The four typing approaches depicted in Figure 13-1 are complementary: they have different pros and cons. It doesn't make sense to dismiss any of them.



Runtime checking
Duck typing
Goose typing
Any Python, avoiding isinstance checks
Python ≥ 2.6 using isinstance checks against ABCs
Structural types
Nominal types
Python ≥ 3.8 with PEP 544 typing.Protocol type hints and external type checker
Python ≥ 3.5 with PEP 484 type hints and external type checker
Static duck typing
Static typing
Static checking

Figure 13-1. The top half describes runtime type checking approaches using just the Python interpreter; the bottom requires an external static type checker such as MyPy or an IDE like PyCharm. The left quadrants cover typing based on the object's structure—i.e., the methods provided by the object, regardless of the name of its class or superclasses; the right quadrants depend on objects having explicitly named types: the name of the object's class, or the name of its superclasses.

Each of these four approaches rely on interfaces to work, but static typing can be done—poorly—using only concrete types instead of interface abstractions like protocols and abstract base classes. This chapter is about duck typing, goose typing, and static duck typing—typing disciplines that revolve around interfaces.

This chapter is split in four main sections, addressing three of the four quadrants in the Typing Map (Figure 13-1):

- "Two Kinds of Protocols" on page 434 compares the two forms of structural typing with protocols—i.e., the lefthand side of the Typing Map.
- "Programming Ducks" on page 435 dives deeper into Python's usual duck typing, including how to make it safer while preserving its major strength: flexibility.
- "Goose Typing" on page 442 explains the use of ABCs for stricter runtime type checking. This is the longest section, not because it's more important, but because there are more sections about duck typing, static duck typing, and static typing elsewhere in the book.
- "Static Protocols" on page 466 covers usage, implementation, and design of `typing.Protocol` subclasses—useful for static and runtime type checking.

What's New in This Chapter

This chapter was heavily edited and is about 24% longer than the corresponding Chapter 11 in the first edition of *Fluent Python*. Although some sections and many paragraphs are the same, there's a lot of new content. These are the highlights:

- The chapter introduction and the Typing Map (Figure 13-1) are new. That's the key to most new content in this chapter—and all other chapters related to typing in Python ≥ 3.8.
- "Two Kinds of Protocols" on page 434 explains the similarities and differences between dynamic and static protocols.
- "Defensive Programming and 'Fail Fast'" on page 440 mostly reproduces content from the first edition, but was updated and now has a section title to highlight its importance.
- "Static Protocols" on page 466 is all new. It builds on the initial presentation in "Static Protocols" on page 286 (Chapter 8).
- Updated class diagrams of `collections.abc` in Figures 13-2, 13-3, and 13-4 to include the `Collection` ABC, from Python 3.6.

The first edition of *Fluent Python* had a section encouraging use of the `numbers` ABCs for goose typing. In "The numbers ABCs and Numeric Protocols" on page 478, I explain why you should use numeric static protocols from the `typing` module instead, if you plan to use static type checkers as well as runtime checks in the style of goose typing.

Two Kinds of Protocols

The word *protocol* has different meanings in computer science depending on context. A network protocol such as HTTP specifies commands that a client can send to a server, such as GET, PUT, and HEAD. We saw in "Protocols and Duck Typing" on page 402 that an object protocol specifies methods which an object must provide to fulfill a role. The FrenchDeck example in Chapter 1 demonstrated one object protocol, the sequence protocol: the methods that allow a Python object to behave as a sequence.

Implementing a full protocol may require several methods, but often it is OK to implement only part of it. Consider the Vowels class in Example 13-1.

Example 13-1. Partial sequence protocol implementation with `__getitem__`

```
>>> class Vowels:
...     def __getitem__(self, i):
...         return 'AEIOU'[i]
...
>>> v = Vowels()
>>> v[0]
'A'
>>> v[-1]
'U'
>>> for c in v: print(c)
...
A
E
I
O
U
>>> 'E' in v
True
>>> 'Z' in v
False
```

Implementing `__getitem__` is enough to allow retrieving items by index, and also to support iteration and the `in` operator. The `__getitem__` special method is really the key to the sequence protocol. Take a look at this entry from the *Python/C API Reference Manual*, "Sequence Protocol" section:

`int PySequence_Check(PyObject *o)`

Return `1` if the object provides sequence protocol, and `0` otherwise. Note that it returns `1` for Python classes with a `__getitem__()` method unless they are `dict` subclasses [...].

We expect a sequence to also support `len()`, by implementing `__len__`. `Vowels` has no `__len__` method, but it still behaves as a sequence in some contexts. And that may be enough for our purposes. That is why I like to say that a protocol is an "informal interface." That is also how protocols are understood in Smalltalk, the first object-oriented programming environment to use that term.

Except in pages about network programming, most uses of the word "protocol" in the Python documentation refer to these informal interfaces.

Now, with the adoption of PEP 544—Protocols: Structural subtyping (static duck typing) in Python 3.8, the word "protocol" has another meaning in Python—closely related, but different. As we saw in "Static Protocols" on page 286 (Chapter 8), PEP 544 allows us to create subclasses of `typing.Protocol` to define one or more methods that a class must implement (or inherit) to satisfy a static type checker.

When I need to be specific, I will adopt these terms:

Dynamic protocol
: The informal protocols Python always had. Dynamic protocols are implicit, defined by convention, and described in the documentation. Python's most important dynamic protocols are supported by the interpreter itself, and are documented in the "Data Model" chapter of *The Python Language Reference*.

Static protocol
: A protocol as defined by PEP 544—Protocols: Structural subtyping (static duck typing), since Python 3.8. A static protocol has an explicit definition: a `typing.Protocol` subclass.

There are two key differences between them:

- An object may implement only part of a dynamic protocol and still be useful; but to fulfill a static protocol, the object must provide every method declared in the protocol class, even if your program doesn't need them all.
- Static protocols can be verified by static type checkers, but dynamic protocols can't.

Both kinds of protocols share the essential characteristic that a class never needs to declare that it supports a protocol by name, i.e., by inheritance.

In addition to static protocols, Python provides another way of defining an explicit interface in code: an abstract base class (ABC).

The rest of this chapter covers dynamic and static protocols, as well as ABCs.

Programming Ducks

Let's start our discussion of dynamic protocols with two of the most important in Python: the sequence and iterable protocols. The interpreter goes out of its way to handle objects that provide even a minimal implementation of those protocols, as the next section explains.

Python Digs Sequences

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

Figure 13-2 shows how the `Sequence` interface is formalized as an ABC. The Python interpreter and built-in sequences like `list`, `str`, etc., do not rely on that ABC at all. I am using it only to describe what a full-fledged `Sequence` is expected to support.

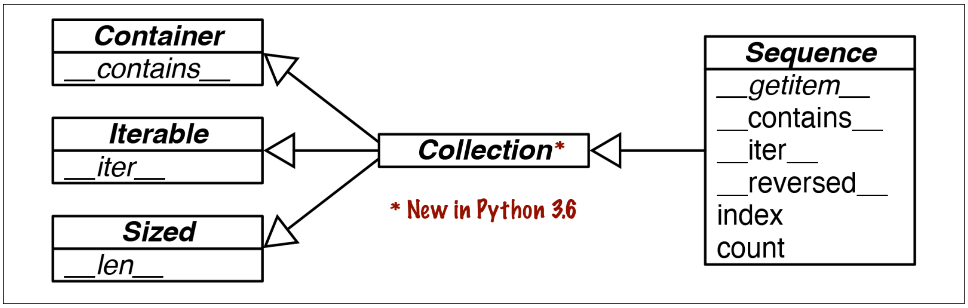


Figure 13-2. UML class diagram for the `Sequence` ABC and related abstract classes from `collections.abc`. Inheritance arrows point from a subclass to its superclasses. Names in italic are abstract methods. Before Python 3.6, there was no `Collection` ABC—`Sequence` was a direct subclass of `Container`, `Iterable`, and `Sized`.

Most ABCs in the `collections.abc` module exist to formalize interfaces that are implemented by built-in objects and are implicitly supported by the interpreter—both of which predate the ABCs themselves. The ABCs are useful as starting points for new classes, and to support explicit type checking at runtime (a.k.a. *goose typing*) as well as type hints for static type checkers.

Studying Figure 13-2, we see that a correct subclass of `Sequence` must implement `__getitem__` and `__len__` (from `Sized`). All the other methods in `Sequence` are concrete, so subclasses can inherit their implementations—or provide better ones.

Now, recall the `Vowels` class in Example 13-1. It does not inherit from `abc.Sequence` and it only implements `__getitem__`.

There is no `__iter__` method, yet `Vowels` instances are iterable because—as a fallback—if Python finds 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 `Vowels` instances, it can also make the `in` operator work even when the `__contains__` method is missing: it does a sequential scan to check if an item is present.

In summary, given the importance of sequence-like data structures, Python manages to make iteration and the `in` operator work by invoking `__getitem__` when `__iter__` and `__contains__` are unavailable.

The original `FrenchDeck` from Chapter 1 does not subclass `abc.Sequence` either, but it does implement both methods of the sequence protocol: `__getitem__` and `__len__`. See Example 13-2.

Example 13-2. A deck as a sequence of cards (same as Example 1-1)

```python
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]
```

Several of the examples in Chapter 1 work because of the special treatment Python gives to anything vaguely resembling a sequence. The iterable protocol in Python represents an extreme form of duck typing: the interpreter tries two different methods to iterate over objects.

To be clear, the behaviors I described in this section are implemented in the interpreter itself, mostly in C. They do not depend on methods from the `Sequence` ABC. For example, the concrete methods `__iter__` and `__contains__` in the `Sequence` class emulate the built-in behaviors of the Python interpreter. If you are curious, check the source code of these methods in *Lib/_collections_abc.py*.

Now let's study another example emphasizing the dynamic nature of protocols—and why static type checkers have no chance of dealing with them.

Monkey Patching: Implementing a Protocol at Runtime

Monkey patching is dynamically changing a module, class, or function at runtime, to add features or fix bugs. For example, the gevent networking library monkey patches parts of Python's standard library to allow lightweight concurrency without threads or `async/await`.[2]

The `FrenchDeck` class from Example 13-2 is missing an essential feature: 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."

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]
```

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

However, if we try to shuffle a `FrenchDeck` instance, we get an exception, as in Example 13-3.

Example 13-3. `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 ".../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 clear: `'FrenchDeck' object does not support item assignment`. The problem is that `shuffle` operates *in place*, by swapping items inside the

2 The "Monkey patch" article on Wikipedia has a funny example in Python.

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 13-4 shows how to do it.

Example 13-4. Monkey patching `FrenchDeck` to make it mutable and compatible with `random.shuffle` (continuing from Example 13-3)

```
>>> 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 shuffled because I added 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 I 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 attributes.

Besides being an example of monkey patching, Example 13-4 highlights the dynamic nature of protocols in dynamic duck typing: `random.shuffle` doesn't care about the class of the argument, it only needs the object to implement methods from 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.

Duck typing doesn't need to be wildly unsafe or hard to debug. The next section shows some useful code patterns to detect dynamic protocols without resorting to explicit checks.

Defensive Programming and "Fail Fast"

Defensive programming is like defensive driving: a set of practices to enhance safety even when faced with careless programmers—or drivers.

Many bugs cannot be caught except at runtime—even in mainstream statically typed languages.[3] In a dynamically typed language, "fail fast" is excellent advice for safer and easier-to-maintain programs. Failing fast means raising runtime errors as soon as possible, for example, rejecting invalid arguments right a the beginning of a function body.

Here is one example: when you write code that accepts a sequence of items to process internally as a `list`, don't enforce a `list` argument by type checking. Instead, take the argument and immediately build a `list` from it. One example of this code pattern is the `__init__` method in Example 13-10, later in this chapter:

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

That way you make your code more flexible, because the `list()` constructor handles any iterable that fits in memory. If the argument is not iterable, the call will fail fast with a very clear `TypeError` exception, right when the object is initialized. If you want to be more explict, you can wrap the `list()` call with `try/except` to customize the error message—but I'd use that extra code only on an external API, because the problem would be easy to see for maintainers of the codebase. Either way, the offending call will appear near the end of the traceback, making it straightforward to fix. If you don't catch the invalid argument in the class constructor, the program will blow up later, when some other method of the class needs to operate on `self._balls` and it is not a `list`. Then the root cause will be harder to find.

Of course, calling `list()` on the argument would be bad if the data shouldn't be copied, either because it's too large or because the function, by design, needs to change it in place for the benefit of the caller, like `random.shuffle` does. In that case, a runtime check like `isinstance(x, abc.MutableSequence)` would be the way to go.

If you are afraid to get an infinite generator—not a common issue—you can begin by calling `len()` on the argument. This would reject iterators, while safely dealing with tuples, arrays, and other existing or future classes that fully implement the `Sequence`

3 That's why automated testing is necessary.

interface. Calling `len()` is usually very cheap, and an invalid argument will raise an error immediately.

On the other hand, if any iterable is acceptable, then call `iter(x)` as soon as possible to obtain an iterator, as we'll see in "Why Sequences Are Iterable: The iter Function" on page 596. Again, if `x` is not iterable, this will fail fast with an easy-to-debug exception.

In the cases I just described, a type hint could catch some problems earlier, but not all problems. Recall that the type `Any` is *consistent-with* every other type. Type inference may cause a variable to be tagged with the `Any` type. When that happens, the type checker is in the dark. In addition, type hints are not enforced at runtime. Fail fast is the last line of defense.

Defensive code leveraging duck types can also include logic to handle different types without using `isinstance()` or `hasattr()` tests.

One example is how we might emulate 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. Example 13-5 shows how I'd do it using duck typing.

Example 13-5. Duck typing to handle a string or an iterable of strings

```
    try:  ➊
        field_names = field_names.replace(',', ' ').split()  ➋
    except AttributeError:  ➌
        pass  ➍
    field_names = tuple(field_names)  ➎
    if not all(s.isidentifier() for s in field_names):  ➏
        raise ValueError('field_names must all be valid identifiers')
```

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

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

➌ Sorry, `field_names` doesn't quack like a `str`: it has no `.replace`, or it returns something we can't `.split`.

➍ If `AttributeError` was raised, then `field_names` is not a `str` and we assume it was already an iterable of names.

➎ To make sure it's an iterable and to keep our own copy, create a tuple out of what we have. A `tuple` is more compact than `list`, and it also prevents my code from changing the names by mistake.

❻ Use `str.isidentifier` to ensure every name is valid.

Example 13-5 shows one situation where duck typing is more expressive than static type hints. There is no way to spell a type hint that says "`field_names` must be a string of identifiers separated by spaces or commas." This is the relevant part of the `namedtuple` signature on typeshed (see the full source at *stdlib/3/collections/__init__.pyi*):

```
def namedtuple(
    typename: str,
    field_names: Union[str, Iterable[str]],
    *,
    # rest of signature omitted
```

As you can see, `field_names` is annotated as `Union[str, Iterable[str]]`, which is OK as far as it goes, but is not enough to catch all possible problems.

After reviewing dynamic protocols, we move to a more explicit form of runtime type checking: goose typing.

Goose Typing

> An abstract class represents an interface.
>
> —Bjarne Stroustrup, creator of C++[4]

Python doesn't have an `interface` keyword. We use abstract base classes (ABCs) to define interfaces for explicit type checking at runtime—also supported by static type checkers.

The *Python Glossary* entry for abstract base class has a good explanation of the value they bring to duck-typed languages:

> Abstract base classes complement duck typing by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example, with magic methods). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation.[5]

Goose typing is a runtime type checking approach that leverages ABCs. I will let Alex Martelli explain in "Waterfowl and ABCs" on page 443.

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

5 Retrieved October 18, 2020.

I am very grateful to my friends Alex Martelli and Anna Ravenscroft. I showed them the first outline of *Fluent Python* at OSCON 2013, and they encouraged me to submit it for publication with O'Reilly. Both later contributed with thorough tech reviews. Alex was already the most cited person in this book, and then he offered to write this essay. 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 object-oriented programming example:

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

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

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

Clearly, the mere existence of a method named `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*).[6]

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*

6 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.*

Effective C++, Addison-Wesley), 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 object-oriented 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, noncompliant 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)
```

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 the "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!*

To summarize, *goose typing* entails:

- Subclassing from ABCs to make it explict that you are implementing a previously defined interface.
- Runtime type checking using ABCs instead of concrete classes as the second argument for `isinstance` and `issubclass`.

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.

Details of using `register` are covered in "A Virtual Subclass of an ABC" on page 460, later in this chapter. For now, here is a brief example: given the `FrenchDeck` class, if I want it to pass a check like `issubclass(FrenchDeck, Sequence)`, I can make it a *virtual subclass* of the `Sequence` ABC with these lines:

```
from collections.abc import Sequence
Sequence.register(FrenchDeck)
```

The use of `isinstance` and `issubclass` becomes more acceptable if you are checking against ABCs instead of concrete classes. If used with concrete classes, type checks limit polymorphism—an essential feature of object-oriented programming. But with ABCs these tests are more flexible. After all, if a component does not implement an ABC by subclassing—but does implement the required methods—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 `isinstance` checks performing different actions depending on the type of object: you should be using polymorphism for that—i.e., design your classes so that the interpreter dispatches calls to the proper methods, instead of you hardcoding the dispatch logic in `if/elif/elif` blocks.

On the other hand, it's OK to perform an `isinstance` 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.

Finally, in his essay, Alex reinforces more than once the need for restraint in the creation of ABCs. Excessive use of ABCs would impose ceremony in a language that became popular because it is practical and pragmatic. During the *Fluent Python* review process, Alex wrote in an e-mail:

> 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.

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 13-6, `FrenchDeck2` is explicitly declared a subclass of `collections.MutableSequence`.

Example 13-6. frenchdeck2.py: `FrenchDeck2`, a subclass of `collections.MutableSequence`

```
from collections import namedtuple, abc

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

class FrenchDeck2(abc.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 of the abstract methods, 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 13-3 shows, not all methods of the `Sequence` and `MutableSequence` ABCs are abstract.

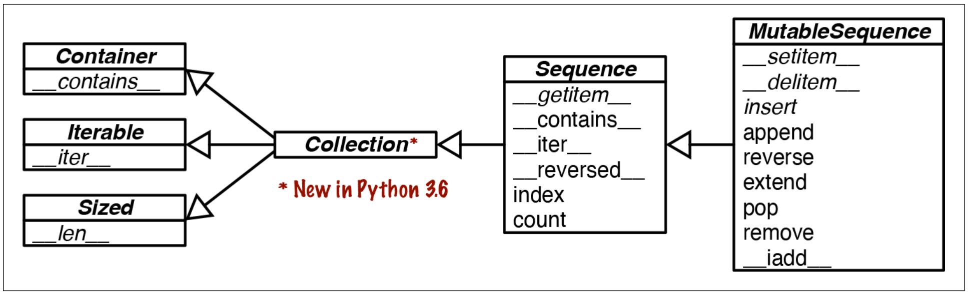


Figure 13-3. UML class diagram for the `MutableSequence` ABC and its superclasses from `collections.abc` (inheritance arrows point from subclasses to ancestors; names in italic are abstract classes and abstract methods).

To write `FrenchDeck2` as a subclass of `MutableSequence`, I had to pay the price of implementing `__delitem__` and `insert`, which my examples did not require. In return, `FrenchDeck2` inherits five concrete methods from `Sequence`: `__contains__`, `__iter__`, `__reversed__`, `index`, and `count`. From `MutableSequence`, it gets another six methods: `append`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`—which supports the += operator for in place concatenation.

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 sequential 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 the `bisect` function from the standard library. See "Managing Ordered Sequences with Bisect" at *fluentpython.com* to learn more about it.

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, the standard library provides several ABCs. Most are defined in the `collections.abc` module, but there are others. You can find ABCs in the `io` and `numbers` packages, for example. But the most widely used are in `collections.abc`.

There are two modules named `abc` in the standard library. Here we are talking about `collections.abc`. To reduce loading time, since Python 3.4 that module is 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 the `abc` module, but we don't need to import it ourselves except to create a brand-new ABC.

Figure 13-4 is a summary UML class diagram (without attribute names) of 17 ABCs defined in `collections.abc`. The 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 13-4. We'll devote most of Chapter 14 to multiple inheritance, but for now it's enough to say that it is usually not a problem when ABCs are concerned.[7]

7 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.

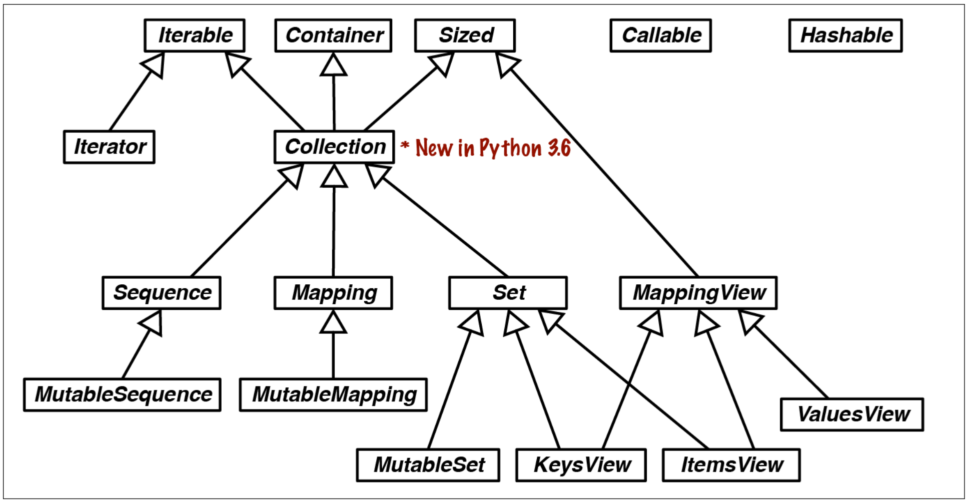


Figure 13-4. UML class diagram for ABCs in `collections.abc`.

Let's review the clusters in Figure 13-4:

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

`Collection`
: This ABC has no methods of its own, but was added in Python 3.6 to make it easier to subclass from `Iterable`, `Container`, and `Sized`.

`Sequence`, `Mapping`, `Set`
: These are the main immutable collection types, and each has a mutable subclass. A detailed diagram for `MutableSequence` is in Figure 13-3; for `MutableMapping` and `MutableSet`, see Figures 3-1 and 3-2 in Chapter 3.

`MappingView`
: In Python 3, the objects returned from the mapping methods `.items()`, `.keys()`, and `.values()` implement the interfaces defined in `ItemsView`, `KeysView`, and `ValuesView`, respectively. The first two also implement the rich interface of `Set`, with all the operators we saw in "Set Operations" on page 107.

`Iterator`
: Note that iterator subclasses `Iterable`. We discuss this further in Chapter 17.

`Callable`, `Hashable`
: These are not 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. They support type checking objects that must be callable or hashable.

For callable detection, the `callable(obj)` built-in function is more convenient than `insinstance(obj, Callable)`.

If `insinstance(obj, Hashable)` returns `False`, you can be certain that `obj` is not hashable. But if the return is `True`, it may be a false positive. The next box explains.

isinstance with Hashable and Iterable Can Be Misleading

It's easy to misinterpret the results of the `isinstance` and `issubclass` tests against the `Hashable` and `Iterable` ABCs.

If `isinstance(obj, Hashable)` returns `True`, that only means that the class of `obj` implements or inherits `__hash__`. But if `obj` is a `tuple` containing unhashable items, then `obj` is not hashable, despite the positive result of the `isinstance` check. Tech reviewer Jürgen Gmach pointed out that duck typing provides the most accurate way to determine if an instance is hashable: call `hash(obj)`. That call will raise `TypeError` if `obj` is not hashable.

On the other hand, even when `isinstance(obj, Iterable)` returns `False`, Python may still be able to iterate over `obj` using `__getitem__` with 0-based indices, as we saw in Chapter 1 and "Python Digs Sequences" on page 436. The documentation for `collections.abc.Iterable` states:

> The only reliable way to determine whether an object is iterable is to call `iter(obj)`.

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 creating 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

This warning appeared in the "Interfaces" chapter of the first edition of *Fluent Python*:

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

Now ABCs have more potential use cases in type hints to support static typing. As discussed in "Abstract Base Classes" on page 278, using ABCs instead of concrete types in function argument type hints gives more flexibility to the caller.

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.

In the literature about data structures, "stack" and "queue" describe abstract interfaces in terms of physical arrangements of objects. I will follow suit and 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.

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 `tuple` built from the items currently in the container, without changing its contents (the internal ordering is not preserved).

Figure 13-5 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…

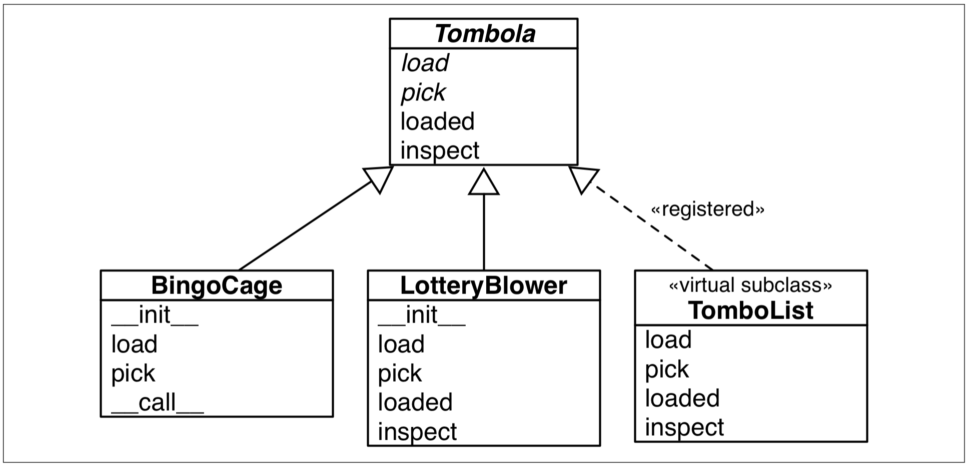


Figure 13-5. UML diagram for an ABC and three subclasses. The name of the `Tombola` ABC and its abstract methods are written in italics, per UML conventions. The dashed arrow is used for interface implementation—here I am using it to show that `TomboList` not only implements the `Tombola` interface, but is also registered as virtual subclass of `Tombola`—as we will see later in this chapter.[9]

Example 13-7 shows the definition of the `Tombola` ABC.

Example 13-7. 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.
        """

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

9 «registered» and «virtual subclass» are not standard UML terms. I am using them to represent a class relationship that is specific to Python.

```
        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(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.[10]

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.

10 Before ABCs existed, abstract methods would raise `NotImplementedError` to signal that subclasses were responsible for their implementation. In Smalltalk-80, abstract method bodies would invoke `subclassResponsibility`, a method inherited from `object` that would produce an error with the message, "My subclass should have overridden one of my messages."

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 code for the `.inspect()` method in Example 13-7 is silly, but it shows that we can rely on `.pick()` and `.load(…)` to inspect what's inside the `Tombola` by picking all items and loading them back—without knowing how the items are actually stored. 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 `Tombola` may always override `.inspect()` with a smarter implementation, but they don't have to.

The `.loaded()` method in Example 13-7 has one line, but it's expensive: it calls `.inspect()` to build the `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 make this explicit in Python, except in the documentation (see the docstring for the abstract `pick` method in Example 13-7).

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`, `KeyError`, or a custom subclass of `LookupError` to comply. See Figure 13-6.

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

Figure 13-6. Part of the `Exception` class hierarchy.[11]

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 13-8.

Example 13-8. 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'>
>>> f = Fake()  ❸
Traceback (most recent call last):
```

11 The complete tree is in section "5.4. Exception hierarchy" of *The Python Standard Library* docs.

```
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Fake with abstract method 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 standard way to declare an ABC is to subclass `abc.ABC` or any other ABC.

Besides the `ABC` base class, and the `@abstractmethod` decorator, the `abc` module defines the `@abstractclassmethod`, `@abstractstaticmethod`, and `@abstractproperty` decorators. However, these last three were deprecated in 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 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've got these ABC syntax issues covered, let's put `Tombola` to use by implementing two concrete descendants of it.

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

Subclassing an ABC

Given the Tombola ABC, we'll now develop two concrete subclasses that satisfy its interface. These classes were pictured in Figure 13-5, along with the virtual subclass to be discussed in the next section.

The BingoCage class in Example 13-9 is a variation of Example 7-8 using a better randomizer. This BingoCage implements the required abstract methods load and pick.

Example 13-9. bingo.py: BingoCage is a concrete subclass of Tombola

```
import random

from tombola import Tombola


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 7-8.

❻ __call__ is also from Example 7-8. 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 13-10. 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 13-10 shows a very different but equally valid implementation of the Tombola interface. Instead of shuffling the "balls" and popping the last, LottoBlower pops from a random position.

Example 13-10. lotto.py: LottoBlower is a concrete subclass that overrides the inspect and loaded methods from Tombola

```
import random

from tombola import Tombola


class LottoBlower(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 LottoBlower')
        return self._balls.pop(position)  ❸

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

    def inspect(self):  ❺
        return tuple(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 13-7). We can make it faster by working with `self._balls` directly—no need to build a whole new `tuple`.

❺ Override `inspect` with a one-liner.

Example 13-10 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 `self._balls = iterable`, aliasing the argument). As mentioned in "Defensive Programming and 'Fail Fast'" on page 440, this makes our `LottoBlower` 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, which is a good practice considering we will be removing items from it and the client might not expect that the provided list will be changed.[13]

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

A Virtual Subclass of an ABC

An essential characteristic of goose typing—and one 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` class method on the ABC. The registered class then becomes a virtual subclass of the ABC, and will be recognized as such by `issubclass`, but it does not inherit any methods or attributes from the ABC.

13 "Defensive Programming with Mutable Parameters" on page 216 in Chapter 6 was devoted to the aliasing issue we just avoided here.

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. Also, static type checkers can't handle virtual subclasses at this time. For details, see Mypy issue 2922—ABCMeta.register support.

The `register` method is usually invoked as a plain function (see "Usage of register in Practice" on page 463), but it can also be used as a decorator. In Example 13-11, we use the decorator syntax and implement `TomboList`, a virtual subclass of `Tombola`, depicted in Figure 13-7.

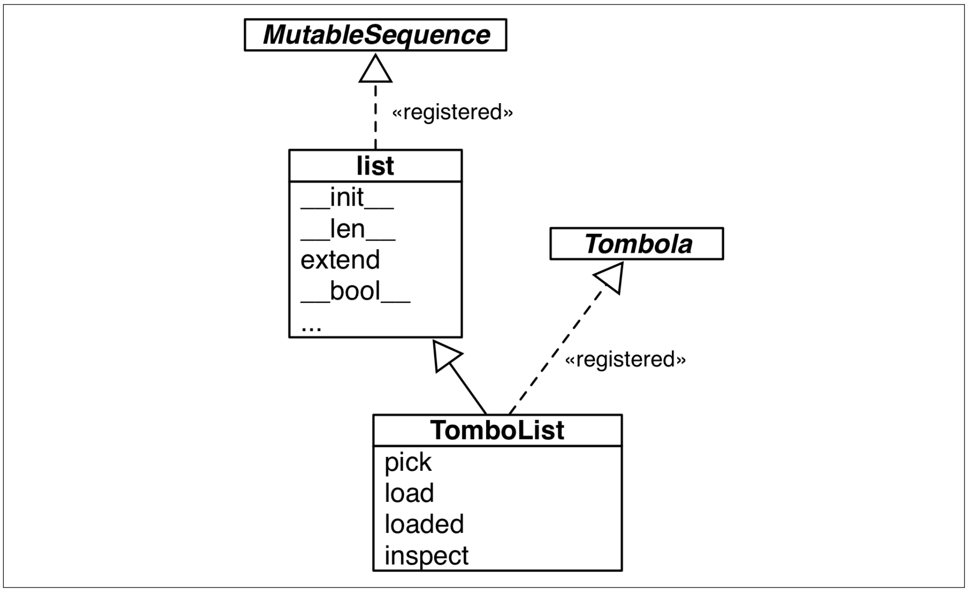


Figure 13-7. UML class diagram for the `TomboList`, a real subclass of `list` and a virtual subclass of `Tombola`.

Example 13-11. 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(self)

# Tombola.register(TomboList)  ❼
```

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

❷ `Tombolist` extends `list`.

❸ `Tombolist` inherits its boolean behavior 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`.[14]

❼ It's always possible to call `register` in this way, and it's useful to do so when you need to register a class that you do not maintain, but which does fulfill the interface.

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
```

14 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`. The `bool()` built-in doesn't need `__bool__` to work because it can also use `__len__`. See "4.1. Truth Value Testing" in the "Built-in Types" chapter of the Python documentation.

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.[15] 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`.

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 13-11, 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 13-11. However, even now, 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. For example, you need to import `MutableMapping` from `collections.abc` to perform a check like `isinstance(my_dict, MutableMapping)`.

Subclassing an ABC or registering with an ABC are both explicit ways of making our classes pass `issubclass` checks—as well as `isinstance` checks, which also rely on `issubclass`. But some ABCs support structural typing as well. The next section explains.

15 There is a whole section explaining the `__mro__` class attribute in "Multiple Inheritance and Method Resolution Order" on page 494. Right now, this quick explanation will do.

Structural Typing with ABCs

ABCs are mostly used with nominal typing. When a class `Sub` explicitly inherits from `AnABC`, or is registered with `AnABC`, the name of `AnABC` is linked to the `Sub` class—and that's how at runtime, `issubclass(AnABC, Sub)` returns `True`.

In contrast, structural typing is about looking at the structure of an object's public interface to determine its type: an object is *consistent-with* a type if it implements the methods defined in the type.[16] Dynamic and static duck typing are two approaches to structural typing.

It turns out that some ABCs also support structural typing. In his essay, "Waterfowl and ABCs" on page 443, Alex shows that a class can be recognized as a 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__`.

The `__subclasshook__` for `Sized` checks whether the class argument has an attribute named `__len__`. If it does, then it is considered a virtual subclass of `Sized`. See Example 13-12.

Example 13-12. Definition of `Sized` from the source code of Lib/_collections_abc.py

```
class Sized(metaclass=ABCMeta):

    __slots__ = ()

    @abstractmethod
    def __len__(self):
        return 0

    @classmethod
    def __subclasshook__(cls, C):
```

16 The concept of type consistency was explained in "Subtype-of versus consistent-with" on page 267.

```
        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 Python 3.6: *Lib/abc.py*. Beware: it has lots of ifs and two recursive calls. In Python 3.7, Ivan Levkivskyi and Inada Naoki rewrote in C most of the logic for the `abc` module, for better performance. See Python issue #31333. The current implementation of `ABCMeta.__subclasscheck__` simply calls `_abc_subclasscheck`. The relevant C source code is in *cpython/Modules/_abc.c#L605*.

That's how `__subclasshook__` allows ABCs to support structural typing. You can formalize an interface with an ABC, you can make `isinstance` checks against that ABC, and still have a completely unrelated class pass an `issubclass` check because it implements a certain method (or because it does whatever it takes to convince a `__subclasshook__` to vouch for it).

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 subtypes of `Sequence`, because you can't retrieve items using integer offsets or slices. That's why the `abc.Sequence` class does not implement `__subclasshook__`.

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 registering it with

`Tombola.register(Spam)`. Of course, your `__subclasshook__` could also check method signatures and other features, but I just don't think it's worthwhile.

Static Protocols

Static protocols were introduced in "Static Protocols" on page 286 (Chapter 8). I considered delaying all coverage of protocols until this chapter, but decided that the initial presentation of type hints in functions had to include protocols because duck typing is an essential part of Python, and static type checking without protocols doesn't handle Pythonic APIs very well.

We will wrap up this chapter by illustrating static protocols with two simple examples, and a discussion of numeric ABCs and protocols. Let's start by showing how a static protocol makes it possible to annotate and type check the `double()` function we first saw in "Types Are Defined by Supported Operations" on page 260.

The Typed double Function

When introducing Python to programmers more used to statically typed languages, one of my favorite examples is this simple `double` function:

```
>>> def double(x):
...     return x * 2
...
>>> double(1.5)
3.0
>>> double('A')
'AA'
>>> double([10, 20, 30])
[10, 20, 30, 10, 20, 30]
>>> from fractions import Fraction
>>> double(Fraction(2, 5))
Fraction(4, 5)
```

Before static protocols were introduced, there was no practical way to add type hints to `double` without limiting its possible uses.[17]

17 OK, `double()` is not very useful, except as an example. But the Python standard library has many functions that could not be properly annotated before static protocols were added in Python 3.8. I helped fix a couple of bugs in *typeshed* by adding type hints using protocols. For example, the pull request that fixed "Should Mypy warn about potential invalid arguments to `max`?" leveraged a `_SupportsLessThan` protocol, which I used to enhance the annotations for `max`, `min`, `sorted`, and `list.sort`.

Thanks to duck typing, `double` works even with types from the future, such as the enhanced `Vector` class that we'll see in "Overloading * for Scalar Multiplication" on page 572 (Chapter 16):

```
>>> from vector_v7 import Vector
>>> double(Vector([11.0, 12.0, 13.0]))
Vector([22.0, 24.0, 26.0])
```

The initial implementation of type hints in Python was a nominal type system: the name of a type in an annotation had to match the name of the type of the actual arguments—or the name of one of its superclasses. Since it's impossible to name all types that implement a protocol by supporting the required operations, duck typing could not be described by type hints before Python 3.8.

Now, with `typing.Protocol` we can tell Mypy that `double` takes an argument `x` that supports `x * 2`. Example 13-13 shows how.

Example 13-13. double_protocol.py: definition of `double` using a `Protocol`

```
from typing import TypeVar, Protocol

T = TypeVar('T')  ❶

class Repeatable(Protocol):
    def __mul__(self: T, repeat_count: int) -> T: ...  ❷

RT = TypeVar('RT', bound=Repeatable)  ❸

def double(x: RT) -> RT:  ❹
    return x * 2
```

❶ We'll use this `T` in the `__mul__` signature.

❷ `__mul__` is the essence of the `Repeatable` protocol. The `self` parameter is usually not annotated—its type is assumed to be the class. Here we use `T` to make sure the result type is the same as the type of `self`. Also, note that `repeat_count` is limited to `int` in this protocol.

❸ The `RT` type variable is bounded by the `Repeatable` protocol: the type checker will require that the actual type implements `Repeatable`.

❹ Now the type checker is able to verify that the `x` parameter is an object that can be multiplied by an integer, and the return value has the same type as `x`.

This example shows why PEP 544 is titled "Protocols: Structural subtyping (static duck typing)." The nominal type of the actual argument `x` given to `double` is irrelevant as long as it quacks—that is, as long as it implements `__mul__`.

Runtime Checkable Static Protocols

In the Typing Map (Figure 13-1), `typing.Protocol` appears in the static checking area—the bottom half of the diagram. However, when defining a `typing.Protocol` subclass, you can use the `@runtime_checkable` decorator to make that protocol support `isinstance/issubclass` checks at runtime. This works because `typing.Protocol` is an ABC, therefore it supports the `__subclasshook__` we saw in "Structural Typing with ABCs" on page 464.

As of Python 3.9, the `typing` module includes seven ready-to-use protocols that are runtime checkable. Here are two of them, quoted directly from the `typing` documentation:

`class typing.SupportsComplex`
  An ABC with one abstract method, `__complex__`.

`class typing.SupportsFloat`
  An ABC with one abstract method, `__float__`.

These protocols are designed to check numeric types for "convertibility": if an object `o` implements `__complex__`, then you should be able to get a `complex` by invoking `complex(o)`—because the `__complex__` special method exists to support the `complex()` built-in function.

Example 13-14 shows the source code for the `typing.SupportsComplex` protocol.

Example 13-14. `typing.SupportsComplex` protocol source code

```python
@runtime_checkable
class SupportsComplex(Protocol):
    """An ABC with one abstract method __complex__."""
    __slots__ = ()

    @abstractmethod
    def __complex__(self) -> complex:
        pass
```

The key is the `__complex__` abstract method.[18] During static type checking, an object will be considered *consistent-with* the `SupportsComplex` protocol if it implements a `__complex__` method that takes only `self` and returns a `complex`.

Thanks to the `@runtime_checkable` class decorator applied to `SupportsComplex`, that protocol can also be used with `isinstance` checks in Example 13-15.

Example 13-15. Using `SupportsComplex` at runtime

```
>>> from typing import SupportsComplex
>>> import numpy as np
>>> c64 = np.complex64(3+4j)  ❶
>>> isinstance(c64, complex)   ❷
False
>>> isinstance(c64, SupportsComplex)  ❸
True
>>> c = complex(c64)  ❹
>>> c
(3+4j)
>>> isinstance(c, SupportsComplex) ❺
False
>>> complex(c)
(3+4j)
```

❶ `complex64` is one of five complex number types provided by NumPy.

❷ None of the NumPy complex types subclass the built-in `complex`.

❸ But NumPy's complex types implement `__complex__`, so they comply with the `SupportsComplex` protocol.

❹ Therefore, you can create built-in `complex` objects from them.

❺ Sadly, the `complex` built-in type does not implement `__complex__`, although `complex(c)` works fine if `c` is a `complex`.

As a result of that last point, if you want to test whether an object `c` is a `complex` or `SupportsComplex`, you can provide a tuple of types as the second argument to `isinstance`, like this:

```
    isinstance(c, (complex, SupportsComplex))
```

18 The `__slots__` attribute is irrelevant to the current discussion—it's an optimization we covered in "Saving Memory with `__slots__`" on page 384.

An alternative would be to use the `Complex` ABC, defined in the `numbers` module. The built-in `complex` type and the NumPy `complex64` and `complex128` types are all registered as virtual subclasses of `numbers.Complex`, therefore this works:

```
>>> import numbers
>>> isinstance(c, numbers.Complex)
True
>>> isinstance(c64, numbers.Complex)
True
```

I recommended using the `numbers` ABCs in the first edition of *Fluent Python*, but now that's no longer good advice, because those ABCs are not recognized by the static type checkers, as we'll see in "The numbers ABCs and Numeric Protocols" on page 478.

In this section I wanted to demonstrate that a runtime checkable protocol works with `isinstance`, but it turns out this is example not a particularly good use case of `isinstance`, as the sidebar "Duck Typing Is Your Friend" on page 470 explains.

If you're using an external type checker, there is one advantage of explicit `isinstance` checks: when you write an `if` statement where the condition is `isinstance(o, MyType)`, then Mypy can infer that inside the `if` block, the type of the `o` object is *consistent-with* `MyType`.

Duck Typing Is Your Friend

Very often at runtime, duck typing is the best approach for type checking: instead of calling `isinstance` or `hasattr`, just try the operations you need to do on the object, and handle exceptions as needed. Here is a concrete example.

Continuing the previous discussion—given an object `o` that I need to use as a complex number, this would be one approach:

```
if isinstance(o, (complex, SupportsComplex)):
    # do something that requires `o` to be convertible to complex
else:
    raise TypeError('o must be convertible to complex')
```

The goose typing approach would be to use the `numbers.Complex` ABC:

```
if isinstance(o, numbers.Complex):
    # do something with `o`, an instance of `Complex`
else:
    raise TypeError('o must be an instance of Complex')
```

However, I prefer to leverage duck typing and do this using the EAFP principle—it's easier to ask for forgiveness than permission:

```
    try:
        c = complex(o)
    except TypeError as exc:
        raise TypeError('o must be convertible to complex') from exc
```

And, if all you're going to do is raise a `TypeError` anyway, then I'd omit the `try/except/raise` statements and just write this:

```
    c = complex(o)
```

In this last case, if `o` is not an acceptable type, Python will raise an exception with a very clear message. For example, this is what I get if `o` is a `tuple`:

```
    TypeError: complex() first argument must be a string or a number, not 'tuple'
```

I find the duck typing approach much better in this case.

Now that we've seen how to use static protocols at runtime with preexisting types like `complex` and `numpy.complex64`, we need to discuss the limitations of runtime checkable protocols.

Limitations of Runtime Protocol Checks

We've seen that type hints are generally ignored at runtime, and this also affects the use of `isinstance` or `issubclass` checks against static protocols.

For example, any class with a `__float__` method is considered—at runtime—a virtual subclass of `SupportsFloat`, even if the `__float__` method does not return a `float`.

Check out this console session:

```
>>> import sys
>>> sys.version
'3.9.5 (v3.9.5:0a7dcbdb13, May  3 2021, 13:17:02) \n[Clang 6.0 (clang-600.0.57)]'
>>> c = 3+4j
>>> c.__float__
<method-wrapper '__float__' of complex object at 0x10a16c590>
>>> c.__float__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert complex to float
```

In Python 3.9, the `complex` type does have a `__float__` method, but it exists only to raise a `TypeError` with an explicit error message. If that `__float__` method had annotations, the return type would be `NoReturn`—which we saw in "NoReturn" on page 294.

But type hinting `complex.__float__` on *typeshed* would not solve this problem because Python's runtime generally ignores type hints—and can't access the *typeshed* stub files anyway.

Continuing from the previous Python 3.9 session:

```
>>> from typing import SupportsFloat
>>> c = 3+4j
>>> isinstance(c, SupportsFloat)
True
>>> issubclass(complex, SupportsFloat)
True
```

So we have misleading results: the runtime checks against `SupportsFloat` suggest that you can convert a `complex` to `float`, but in fact that raises a type error.

The specific isssue with the `complex` type is fixed in Python 3.10.0b4 with the removal of the `complex.__float__` method.

But the overall issue remains: `isinstance`/`issubclass` checks only look at the presence or absence of methods, without checking their signatures, much less their type annotations. And this is not about to change, because such type checks at runtime would have an unacceptable performance cost.[19]

Now let's see how to implement a static protocol in a user-defined class.

Supporting a Static Protocol

Recall the `Vector2d` class we built in Chapter 11. Given that a `complex` number and a `Vector2d` instance both consist of a pair of floats, it makes sense to support conversion from `Vector2d` to `complex`.

Example 13-16 shows the implementation of the `__complex__` method to enhance the last version of `Vector2d` we saw in Example 11-11. For completeness, we can support the inverse operation with a `fromcomplex` class method to build a `Vector2d` from a `complex`.

19 Thanks to Ivan Levkivskyi, coauthor of PEP 544 (on Protocols), for pointing out that type checking is not just a matter of checking whether the type of `x` is `T`: it's about determining that the type of `x` is *consistent-with* `T`, which may be expensive. It's no wonder that Mypy takes a few seconds to type check even short Python scripts.

Example 13-16. vector2d_v4.py: methods for converting to and from `complex`

```
    def __complex__(self):
        return complex(self.x, self.y)

    @classmethod
    def fromcomplex(cls, datum):
        return cls(datum.real, datum.imag)  ❶
```

❶ This assumes that `datum` has `.real` and `.imag` attributes. We'll see a better implementation in Example 13-17.

Given the preceding code, and the `__abs__` method the `Vector2d` already had in Example 11-11, we get these features:

```
>>> from typing import SupportsComplex, SupportsAbs
>>> from vector2d_v4 import Vector2d
>>> v = Vector2d(3, 4)
>>> isinstance(v, SupportsComplex)
True
>>> isinstance(v, SupportsAbs)
True
>>> complex(v)
(3+4j)
>>> abs(v)
5.0
>>> Vector2d.fromcomplex(3+4j)
Vector2d(3.0, 4.0)
```

For runtime type checking, Example 13-16 is fine, but for better static coverage and error reporting with Mypy, the `__abs__`, `__complex__`, and `fromcomplex` methods should get type hints, as shown in Example 13-17.

Example 13-17. vector2d_v5.py: adding annotations to the methods under study

```
    def __abs__(self) -> float:  ❶
        return math.hypot(self.x, self.y)

    def __complex__(self) -> complex:  ❷
        return complex(self.x, self.y)

    @classmethod
    def fromcomplex(cls, datum: SupportsComplex) -> Vector2d:  ❸
        c = complex(datum)  ❹
        return cls(c.real, c.imag)
```

❶ The `float` return annotation is needed, otherwise Mypy infers `Any`, and doesn't check the body of the method.

❷ Even without the annotation, Mypy was able to infer that this returns a `complex`. The annotation prevents a warning, depending on your Mypy configuration.

❸ Here `SupportsComplex` ensures the `datum` is convertible.

❹ This explicit conversion is necessary, because the `SupportsComplex` type does not declare `.real` and `.imag` attributes, used in the next line. For example, `Vector2d` doesn't have those attributes, but implements `__complex__`.

The return type of `fromcomplex` can be `Vector2d` if `from __future__ import annotations` appears at the top of the module. That import causes type hints to be stored as strings, without being evaluated at import time, when function definitions are evaluated. Without the `__future__` import of `annotations`, `Vector2d` is an invalid reference at this point (the class is not fully defined yet) and should be written as a string: `'Vector2d'`, as if it were a forward reference. This `__future__` import was introduced in PEP 563—Postponed Evaluation of Annotations, implemented in Python 3.7. That behavior was scheduled to become default in 3.10, but the change was delayed to a later version.[20] When that happens, the import will be redundant, but harmless.

Next, let's see how to create—and later, extend—a new static protocol.

Designing a Static Protocol

While studying goose typing, we saw the `Tombola` ABC in "Defining and Using an ABC" on page 451. Here we'll see how to define a similar interface using a static protocol.

The `Tombola` ABC specifies two methods: `pick` and `load`. We could define a static protocol with these two methods as well, but I learned from the Go community that single-method protocols make static duck typing more useful and flexible. The Go standard library has several interfaces like `Reader`, an interface for I/O that requires just a `read` method. After a while, if you realize a more complete protocol is required, you can combine two or more protocols to define a new one.

Using a container that picks items at random may or may not require reloading the container, but it certainly needs a method to do the actual pick, so that's the method I

20 Read the Python Steering Council decision on python-dev.

will choose for the minimal RandomPicker protocol. The code for that protocol is in Example 13-18, and its use is demonstrated by tests in Example 13-19.

Example 13-18. randompick.py: definition of RandomPicker

```
from typing import Protocol, runtime_checkable, Any

@runtime_checkable
class RandomPicker(Protocol):
    def pick(self) -> Any: ...
```

The pick method returns Any. In "Implementing a Generic Static Protocol" on page 552, we will see how to make RandomPicker a generic type with a parameter to let users of the protocol specify the return type of the pick method.

Example 13-19. randompick_test.py: RandomPicker in use

```
import random
from typing import Any, Iterable, TYPE_CHECKING

from randompick import RandomPicker  ❶

class SimplePicker:  ❷
    def __init__(self, items: Iterable) -> None:
        self._items = list(items)
        random.shuffle(self._items)

    def pick(self) -> Any:  ❸
        return self._items.pop()

def test_isinstance() -> None:  ❹
    popper: RandomPicker = SimplePicker([1])  ❺
    assert isinstance(popper, RandomPicker)  ❻

def test_item_type() -> None:  ❼
    items = [1, 2]
    popper = SimplePicker(items)
    item = popper.pick()
    assert item in items
    if TYPE_CHECKING:
        reveal_type(item)  ❽
    assert isinstance(item, int)
```

❶ It's not necessary to import the static protocol to define a class that implements it. Here I imported `RandomPicker` only to use it on `test_isinstance` later.

❷ `SimplePicker` implements `RandomPicker`—but it does not subclass it. This is static duck typing in action.

❸ `Any` is the default return type, so this annotation is not strictly necessary, but it does make it more clear that we are implementing the `RandomPicker` protocol as defined in Example 13-18.

❹ Don't forget to add `-> None` hints to your tests if you want Mypy to look at them.

❺ I added a type hint for the `popper` variable to show that Mypy understands that `SimplePicker` is *consistent-with*.

❻ This test proves that an instance of `SimplePicker` is also an instance of `RandomPicker`. This works because of the `@runtime_checkable` decorator applied to `RandomPicker`, and because `SimplePicker` has a `pick` method as required.

❼ This test invokes the `pick` method from a `SimplePicker`, verifies that it returns one of the items given to `SimplePicker`, and then does static and runtime checks on the returned item.

❽ This line generates a note in the Mypy output.

As we saw in Example 8-22, `reveal_type` is a "magic" function recognized by Mypy. That's why it is not imported and we can only call it inside `if` blocks protected by `typing.TYPE_CHECKING`, which is only `True` in the eyes of a static type checker, but is `False` at runtime.

Both tests in Example 13-19 pass. Mypy does not see any errors in that code either, and shows the result of the `reveal_type` on the `item` returned by `pick`:

```
$ mypy randompick_test.py
randompick_test.py:24: note: Revealed type is 'Any'
```

Having created our first protocol, let's study some advice on the matter.

Best Practices for Protocol Design

After 10 years of experience with static duck typing in Go, it is clear that narrow protocols are more useful—often such protocols have a single method, rarely more than a couple of methods. Martin Fowler wrote a post defining *role interface*, a useful idea to keep in mind when designing protocols.

Also, sometimes you see a protocol defined near the function that uses it—that is, defined in "client code" instead of being defined in a library. This is makes it easy to create new types to call that function, which is good for extensibility and testing with mocks.

The practices of narrow protocols and client-code protocols both avoid unnecessary tight coupling, in line with the Interface Segregation Principle, which we can summarize as "Clients should not be forced to depend upon interfaces that they do not use."

The page "Contributing to typeshed" recommends this naming convention for static protocols (the following three points are quoted verbatim):

- Use plain names for protocols that represent a clear concept (e.g., `Iterator`, `Container`).
- Use `SupportsX` for protocols that provide callable methods (e.g., `SupportsInt`, `SupportsRead`, `SupportsReadSeek`).[21]
- Use `HasX` for protocols that have readable and/or writable attributes or getter/setter methods (e.g., `HasItems`, `HasFileno`).

The Go standard library has a naming convention that I like: for single method protocols, if the method name is a verb, append "-er" or "-or" to make it a noun. For example, instead of `SupportsRead`, have `Reader`. More examples include `Formatter`, `Animator`, and `Scanner`. For inspiration, see "Go (Golang) Standard Library Interfaces (Selected)" by Asuka Kenji.

One good reason to create minimalistic protocols is the ability to extend them later, if needed. We'll now see that it's not hard to create a derived protocol with an additional method.

Extending a Protocol

As I mentioned at the start of the previous section, Go developers advocate to err on the side of minimalism when defining interfaces—their name for static protocols. Many of the most widely used Go interfaces have a single method.

When practice reveals that a protocol with more methods is useful, instead of adding methods to the original protocol, it's better to derive a new protocol from it. Extending a static protocol in Python has a few caveats, as Example 13-20 shows.

21 Every method is callable, so this guideline doesn't say much. Perhaps "provide one or two methods"? Anyway, it's a guideline, not a strict rule.

Example 13-20. randompickload.py: extending `RandomPicker`

```
from typing import Protocol, runtime_checkable
from randompick import RandomPicker

@runtime_checkable  ❶
class LoadableRandomPicker(RandomPicker, Protocol):  ❷
    def load(self, Iterable) -> None: ...  ❸
```

❶ If you want the derived protocol to be runtime checkable, you must apply the decorator again—its behavior is not inherited.[22]

❷ Every protocol must explicitly name `typing.Protocol` as one of its base classes in addition to the protocol we are extending. This is different from the way inheritance works in Python.[23]

❸ Back to "regular" object-oriented programming: we only need to declare the method that is new in this derived protocol. The `pick` method declaration is inherited from `RandomPicker`.

This concludes the final example of defining and using a static protocol in this chapter.

To wrap the chapter, we'll go over numeric ABCs and their possible replacement with numeric protocols.

The numbers ABCs and Numeric Protocols

As we saw in "The fall of the numeric tower" on page 279, the ABCs in the `numbers` package of the standard library work fine for runtime type checking.

If you need to check for an integer, you can use `isinstance(x, numbers.Integral)` to accept `int`, `bool` (which subclasses `int`) or other integer types that are provided by external libraries that register their types as virtual subclasses of the `numbers` ABCs. For example, NumPy has 21 integer types—as well as several variations of floating-point types registered as `numbers.Real`, and complex numbers with various bit widths registered as `numbers.Complex`.

22 For details and rationale, please see the section about `@runtime_checkable` in PEP 544—Protocols: Structural subtyping (static duck typing).

23 Again, please read "Merging and extending protocols" in PEP 544 for details and rationale.

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 floating-point numbers that are less precise.

Sadly, the numeric tower was not designed for static type checking. The root ABC—numbers.Number—has no methods, so if you declare x: Number, Mypy will not let you do arithmetic or call any methods on x.

If the numbers ABCs are not supported, what are the options?

A good place to look for typing solutions is the *typeshed* project. As part of the Python standard library, the statistics module has a corresponding *statistics.pyi* stub file with type hints for on *typeshed*. There you'll find the following definitions, which are used to annotate several functions:

```
_Number = Union[float, Decimal, Fraction]
_NumberT = TypeVar('_NumberT', float, Decimal, Fraction)
```

That approach is correct, but limited. It does not support numeric types outside of the standard library, which the numbers ABCs do support at runtime—when the numeric types are registered as virtual subclasses.

The current trend is to recommend the numeric protocols provided by the typing module, which we discussed in "Runtime Checkable Static Protocols" on page 468.

Unfortunately, at runtime, the numeric protocols may let you down. As mentioned in "Limitations of Runtime Protocol Checks" on page 471, the complex type in Python 3.9 implements __float__, but the method exists only to raise TypeError with an explicit message: "can't convert complex to float." It implements __int__ as well, for the same reason. The presence of those methods makes isinstance return misleading results in Python 3.9. In Python 3.10, the methods of complex that unconditionally raised TypeError were removed.[24]

On the other hand, NumPy's complex types implement __float__ and __int__ methods that work, only issuing a warning when each of them is used for the first time:

```
>>> import numpy as np
>>> cd = np.cdouble(3+4j)
>>> cd
(3+4j)
>>> float(cd)
```

24 See Issue #41974—Remove complex.__float__, complex.__floordiv__, etc.

```
<stdin>:1: ComplexWarning: Casting complex values to real
discards the imaginary part
3.0
```

The opposite problem also happens: built-ins `complex`, `float`, and `int`, and also `numpy.float16` and `numpy.uint8`, don't have a `__complex__` method, so `isinstance(x, SupportsComplex)` returns `False` for them.[25] The NumPy complex types, such as `np.complex64`, do implement `__complex__` to convert to a built-in `complex`.

However, in practice, the `complex()` built-in constructor handles instances of all these types with no errors or warnings:

```
>>> import numpy as np
>>> from typing import SupportsComplex
>>> sample = [1+0j, np.complex64(1+0j), 1.0, np.float16(1.0), 1, np.uint8(1)]
>>> [isinstance(x, SupportsComplex) for x in sample]
[False, True, False, False, False, False]
>>> [complex(x) for x in sample]
[(1+0j), (1+0j), (1+0j), (1+0j), (1+0j), (1+0j)]
```

This shows that `isinstance` checks against `SupportsComplex` suggest that those conversions to `complex` would fail, but they all succeed. In the typing-sig mailing list, Guido van Rossum pointed out that the built-in `complex` accepts a single argument, and that's why those conversions work.

On the other hand, Mypy accepts arguments of all those six types in a call to a `to_complex()` function defined like this:

```
def to_complex(n: SupportsComplex) -> complex:
    return complex(n)
```

As I write this, NumPy has no type hints, so its number types are all `Any`.[26] On the other hand, Mypy is somehow "aware" that the built-in `int` and `float` can be converted to `complex`, even though on *typeshed* only the built-in `complex` class has a `__complex__` method.[27]

In conclusion, although numeric types should not be hard to type check, the current situation is this: the type hints PEP 484 eschews the numeric tower and implicitly recommends that type checkers hardcode the subtype relationships among built-in `complex`, `float`, and `int`. Mypy does that, and it also pragmatically accepts that `int` and `float` are *consistent-with* `SupportsComplex`, even though they don't implement `__complex__`.

25 I did not test all the other float and integer variants NumPy offers.

26 The NumPy number types are all registered against the appropriate `numbers` ABCs, which Mypy ignores.

27 That's a well-meaning lie on the part of typeshed: as of Python 3.9, the built-in `complex` type does not actually have a `__complex__` method.

I only found unexpected results when using `isinstance` checks with numeric `Supports*` protocols while experimenting with conversions to or from `complex`. If you don't use complex numbers, you can rely on those protocols instead of the `numbers` ABCs.

The main takeaways for this section are:

- The `numbers` ABCs are fine for runtime type checking, but unsuitable for static typing.
- The numeric static protocols `SupportsComplex`, `SupportsFloat`, etc. work well for static typing, but are unreliable for runtime type checking when complex numbers are involved.

We are now ready for a quick review of what we saw in this chapter.

Chapter Summary

The Typing Map (Figure 13-1) is the key to making sense of this chapter. After a brief introduction to the four approaches to typing, we contrasted dynamic and static protocols, which respectively support duck typing and static duck typing. Both kinds of protocols share the essential characteristic that a class is never required to explicitly declare support for any specific protocol. A class supports a protocol simply by implementing the necessary methods.

The next major section was "Programming Ducks" on page 435, where we explored the lengths to which the Python interpreter goes to make the sequence and iterable dynamic protocols work, including partial implementations of both. We then saw how a class can be made to implement a protocol at runtime through the addition of extra methods via monkey patching. The duck typing section ended with hints for defensive programming, including detection of structural types without explicit `isinstance` or `hasattr` checks using `try/except` and failing fast.

After Alex Martelli introduced goose typing in "Waterfowl and ABCs" on page 443, we saw how to subclass existing ABCs, surveyed important ABCs in the standard library, and created an ABC from scratch, which we then implemented by traditional subclassing and by registration. To close this section, we saw how the `__subclasshook__` special method enables ABCs to support structural typing by recognizing unrelated classes that provide methods fulfilling the interface defined in the ABC.

The last major section was "Static Protocols" on page 466, where we resumed coverage of static duck typing, which started in Chapter 8, in "Static Protocols" on page 286. We saw how the `@runtime_checkable` decorator also leverages `__subclasshook__` to support structural typing at runtime—even though the best use of static protocols is with static type checkers, which can take into account type hints to make structural typing more reliable. Next we talked about the design and coding of a static protocol and how to extend it. The chapter ended with "The numbers ABCs and Numeric Protocols" on page 478, which tells the sad story of the derelict state of the numeric tower and a few existing shortcomings of the proposed alternative: the numeric static protocols such as `SupportsFloat` and others added to the `typing` module in Python 3.8.

The main message of this chapter is that we have four complementary ways of programming with interfaces in modern Python, each with different advantages and drawbacks. You are likely to find suitable use cases for each typing scheme in any modern Python codebase of significant size. Rejecting any one of these approaches will make your work as a Python programmer harder than it needs to be.

Having said that, Python achieved widespread popularity while supporting only duck typing. Other popular languages such as JavaScript, PHP, and Ruby, as well as Lisp, Smalltalk, Erlang, and Clojure—not popular but very influential—are all languages that had and still have tremendous impact by leveraging the power and simplicity of duck typing.

Further Reading

For a quick look at typing pros and cons, as well as the importance of `typing.Protocol` for the health of statically checked codebases, I highly recommend Glyph Lefkowitz's post "I Want A New Duck: `typing.Protocol` and the future of duck typing". I also learned a lot from his post "Interfaces and Protocols", comparing `typing.Protocol` and `zope.interface`—an earlier mechanism for defining interfaces in loosely coupled plug-in systems, used by the Plone CMS, the Pyramid web framework, and the Twisted asynchronous programming framework, a project founded by Glyph.[28]

Great books about Python have—almost by definition—great coverage of duck typing. Two of my favorite Python books had updates released after the first edition of *Fluent Python*: *The Quick Python Book*, 3rd ed., (Manning), by Naomi Ceder; and *Python in a Nutshell*, 3rd ed., by Alex Martelli, Anna Ravenscroft, and Steve Holden (O'Reilly).

For a discussion of the pros and cons of dynamic typing, see Guido van Rossum's interview with Bill Venners in "Contracts in Python: A Conversation with Guido van Rossum, Part IV". An insightful and balanced take on this debate is Martin Fowler's post "Dynamic Typing". He also wrote "Role Interface", which I mentioned in "Best Practices for Protocol Design" on page 476. Although it is not about duck typing, that

28 Thanks to tech reviewer Jürgen Gmach for recommending the "Interfaces and Protocols" post.

post is highly relevant for Python protocol design, as he contrasts narrow role interfaces with the broader public interfaces of classes in general.

The Mypy documentation is often the best source of information for anything related to static typing in Python, including static duck typing, addressed in their "Protocols and structural subtyping" chapter.

The remaining references are all about goose typing. Beazley and Jones's *Python Cookbook*, 3rd ed. (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 of declaring ABCs by subclassing from `abc.ABC` (instead, they use the `metaclass` keyword, which we'll only really need in Chapter 24). Apart from this small detail, the recipe covers the major ABC features very well.

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. Hellmann also uses the old style of ABC declaration: `PluginBase(metaclass=abc.ABCMeta)` instead of the simpler `PluginBase(abc.ABC)` available since Python 3.4.

When using ABCs, multiple inheritance is not only common but practically inevitable, because each of the fundamental collection ABCs—`Sequence`, `Mapping`, and `Set`—extends `Collection`, which in turn extends multiple ABCs (see Figure 13-4). Therefore, Chapter 14 is an important follow-up to this one.

PEP 3119—Introducing Abstract Base Classes gives the rationale for ABCs. PEP 3141—A Type Hierarchy for Numbers presents the ABCs of the `numbers` module, but the discussion in the Mypy issue #3186 "int is not a Number?" includes some arguments about why the numeric tower is unsuitable for static type checking. Alex Waygood wrote a comprehensive answer on StackOverflow, discussing ways to annotate numeric types. I'll keep watching Mypy issue #3186 for the next chapters of this saga, hoping for a happy ending that will make static typing and goose typing compatible—as they should be.

Soapbox

The MVP Journey of Python Static Typing

I work for Thoughtworks, a worldwide leader in Agile software development. At Thoughtworks, we often recommend that our clients should aim to create and deploy MVPs: minimal viable products: "a simple version of a product that is given to users in order to validate the key business assumptions," as defined by my colleague Paulo Caroli in "Lean Inception", a post in Martin Fowler's collective blog.

Guido van Rossum and the other core developers who designed and implemented static typing have followed an MVP strategy since 2006. First, PEP 3107—Function Annotations was implemented in Python 3.0 with very limited semantics: just syntax to attach annotations to function parameters and returns. This was done explicitly to allow for experimentation and collect feedback—key benefits of an MVP.

Eight years later, PEP 484—Type Hints was proposed and approved. Its implementation in Python 3.5 required no changes in the language or standard library—except for adding the `typing` module, on which no other part of the standard library depended. PEP 484 supported only nominal types with generics—similar to Java—but with the actual static checking done by external tools. Important features were missing, like variable annotations, generic built-in types, and protocols. Despite those limitations, this typing MVP was valuable enough to attract investment and adoption by companies with very large Python codebases, like Dropbox, Google, and Facebook; as well as support from professional IDEs, like PyCharm, Wing, and VS Code.

PEP 526—Syntax for Variable Annotations was the first evolutionary step that required changes to the interpreter in Python 3.6. More changes to the Python 3.7 interpreter were made to support PEP 563—Postponed Evaluation of Annotations and PEP 560—Core support for typing module and generic types, which allowed built-in and standard library collections to accept generic type hints out of the box in Python 3.9, thanks to PEP 585—Type Hinting Generics In Standard Collections.

During those years, some Python users—including me—were underwhelmed by the typing support. After I learned Go, the lack of static duck typing in Python was incomprehensible, in a language where duck typing had always been a core strength.

But that is the nature of MVPs: they may not satisfy all potential users, but they can be implemented with less effort, and guide further development with feedback from actual usage in the field.

If there is one thing we all learned from Python 3, it's that incremental progress is safer than big-bang releases. I am glad we did not have to wait for Python 4—if it ever comes—to make Python more attractive to large enterprises, where the benefits of static typing outweigh the added complexity.

Typing Approaches in Popular Languages

Figure 13-8 is a variation of the Typing Map (Figure 13-1) with the names of a few popular languages that support each of the typing approaches.

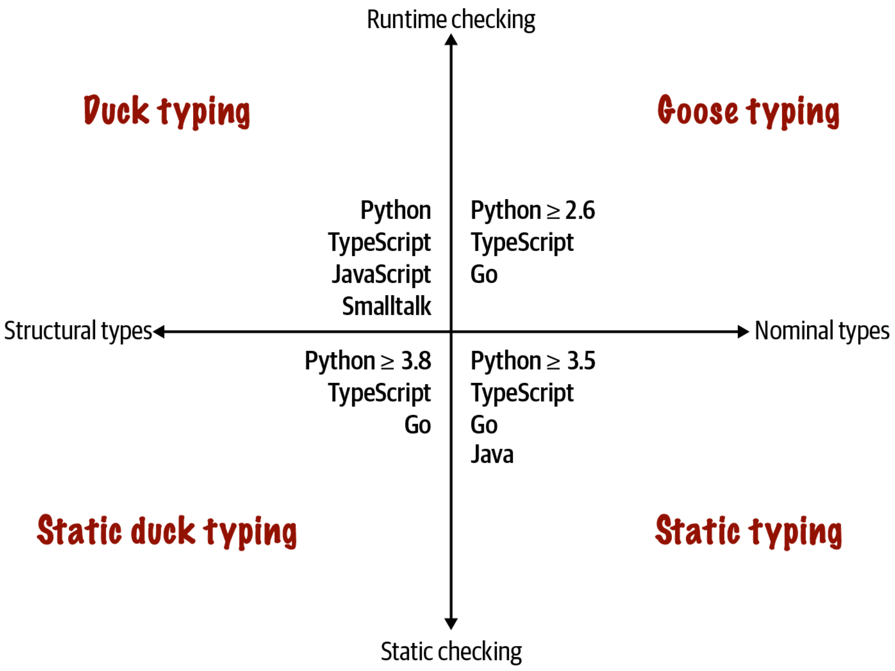


Figure 13-8. Four approaches to type checking and some languages that support them.

TypeScript and Python ≥ 3.8 are the only languages in my small and arbitrary sample that support all four approaches.

Go is clearly a statically typed language in the Pascal tradition, but it pioneered static duck typing—at least among languages that are widely used today. I also put Go in the goose typing quadrant because of its type assertions, which allow checking and adapting to different types at runtime.

If I had to draw a similar diagram in the year 2000, only the duck typing and the static typing quadrants would have languages in them. I am not aware of languages that supported static duck typing or goose typing 20 years ago. The fact that each of the four quadrants has at least three popular languages suggests that a lot of people see value in each of the four approaches to typing.

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 apply conflicting patches.

Metaphors and Idioms in Interfaces

A metaphor fosters understanding by making constraints and affordances clear. That's the value of the words "stack" and "queue" in describing those fundamental data structures: they make clear which operations are allowed, i.e., how items can be added or removed. On the other hand, Alan Cooper et al. write in *About Face, the Essentials of Interaction Design*, 4th ed. (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.

In *About Face*, 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.