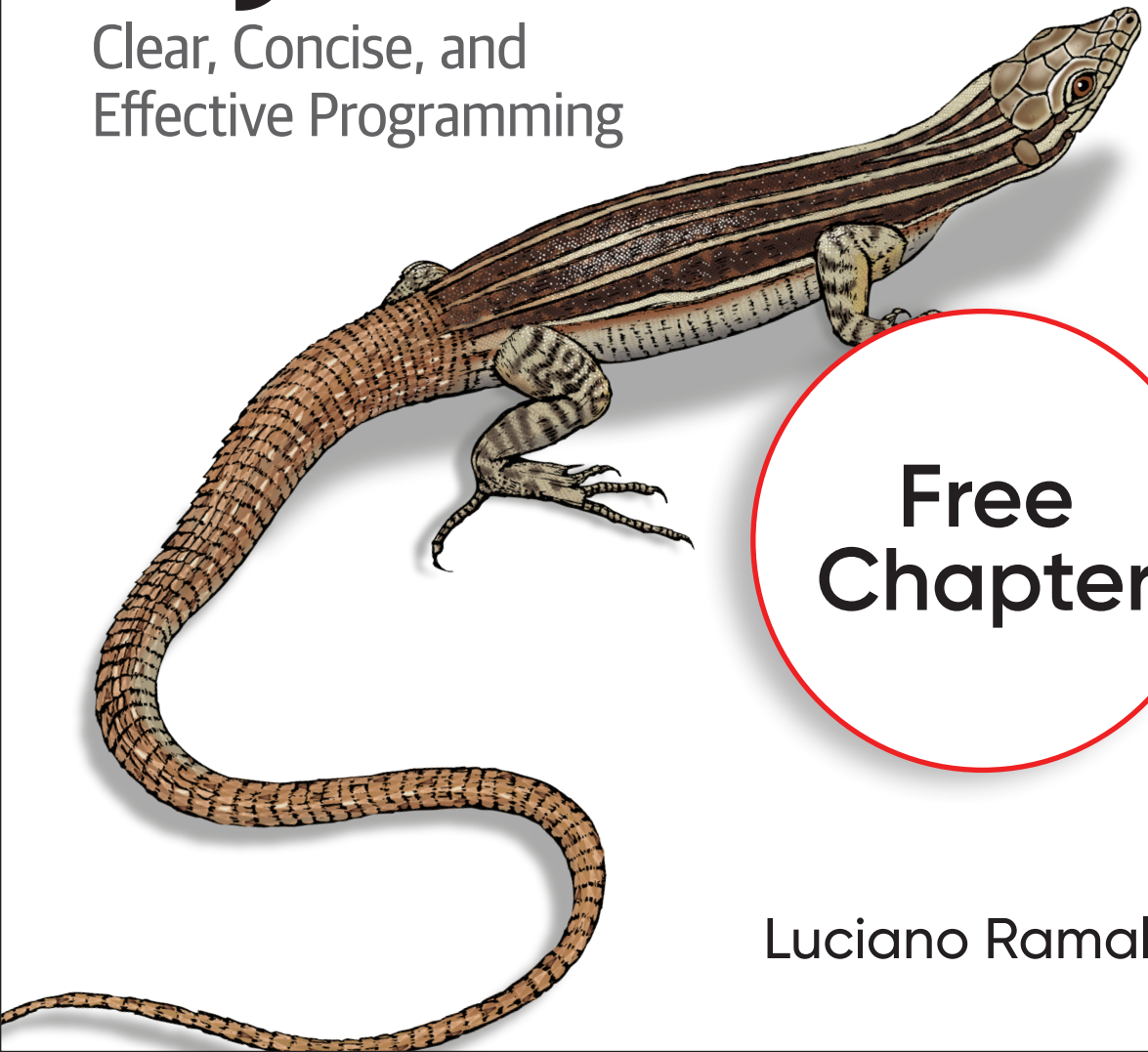
O'REILLY®

Second Edition

Fluent Python

Clear, Concise, and Effective Programming



Free Chapter

Luciano Ramalho

SECOND EDITION

Fluent Python

Clear, Concise, and Effective Programming

This excerpt contains Chapter 1. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Luciano Ramalho

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Fluent Python
by Luciano Ramalho

Copyright © 2022 Luciano Ramalho. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Amanda Quinn
Development Editor: Jeff Bleiel
Production Editor: Daniel Elfanbaum
Copyeditor: Sonia Saruba
Proofreader: Kim Cofer
Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

April 2022: Second Edition

Revision History for the Second Edition
2022-03-31: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492056355* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Fluent Python*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-05635-5

[LSI]

Para Marta, com todo o meu amor.

Table of Contents

CHAPTER 1
The Python Data Model

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

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

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

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

When using a framework, we spend a lot of time coding methods that are called by the framework. The same happens when we leverage the Python Data Model to build new classes. The Python interpreter invokes special methods to perform basic object operations, often triggered by special syntax. The special method names are always written with leading and trailing double underscores. For example, the syntax

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

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

We implement special methods when we want our objects to support and interact with fundamental language constructs such as:

- Collections
- Attribute access
- Iteration (including asynchronous iteration using `async for`)
- Operator overloading
- Function and method invocation
- String representation and formatting
- Asynchronous programming using `await`
- Object creation and destruction
- Managed contexts using the `with` or `async with` statements

Magic and Dunder

The term *magic method* is slang for special method, but how do we talk about a specific method like `__getitem__`? I learned to say "dunder-getitem" from author and teacher Steve Holden. "Dunder" is a shortcut for "double underscore before and after." That's why the special methods are also known as *dunder methods*. The "Lexical Analysis" chapter of *The Python Language Reference* warns that "*Any* use of `__*__` names, in any context, that does not follow explicitly documented use, is subject to breakage without warning."

What's New in This Chapter

This chapter had few changes from the first edition because it is an introduction to the Python Data Model, which is quite stable. The most significant changes are:

- Special methods supporting asynchronous programming and other new features, added to the tables in "Overview of Special Methods" on page 13.
- Figure 1-2 showing the use of special methods in "Collection API" on page 12, including the `collections.abc.Collection` abstract base class introduced in Python 3.6.

Also, here and throughout this second edition I adopted the *f-string* syntax introduced in Python 3.6, which is more readable and often more convenient than the older string formatting notations: the `str.format()` method and the `%` operator.

One reason to still use `my_fmt.format()` is when the definition of `my_fmt` must be in a different place in the code than where the formatting operation needs to happen. For instance, when `my_fmt` has multiple lines and is better defined in a constant, or when it must come from a configuration file, or from the database. Those are real needs, but don't happen very often.

A Pythonic Card Deck

Example 1-1 is simple, but it demonstrates the power of implementing just two special methods, `__getitem__` and `__len__`.

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

```
import collections

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

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

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

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

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

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

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

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

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

Reading specific cards from the deck—say, the first or the last—is easy, thanks to the `__getitem__` method:

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

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

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

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

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

But it gets better.

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

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

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

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

We can also iterate over the deck in reverse:

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

Ellipsis in doctests

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

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

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

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

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

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

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

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

Although `FrenchDeck` implicitly inherits from the `object` class, most of its functionality is not inherited, but comes from leveraging the data model and composition. By implementing the special methods `__len__` and `__getitem__`, our `FrenchDeck` behaves like a standard Python sequence, allowing it to benefit from core language features (e.g., iteration and slicing) and from the standard library, as shown by the examples using `random.choice`, `reversed`, and `sorted`. Thanks to composition, the `__len__` and `__getitem__` implementations can delegate all the work to a `list` object, `self._cards`.

How About Shuffling?

As implemented so far, a `FrenchDeck` cannot be shuffled because it is *immutable*: the cards and their positions cannot be changed, except by violating encapsulation and handling the `_cards` attribute directly. In Chapter 13, we will fix that by adding a one-line `__setitem__` method.

How Special Methods Are Used

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

But the interpreter takes a shortcut when dealing for built-in types like `list`, `str`, `bytearray`, or extensions like the NumPy arrays. Python variable-sized collections written in C include a struct[2] called `PyVarObject`, which has an `ob_size` field holding the number of items in the collection. So, if `my_object` is an instance of one of those built-ins, then `len(my_object)` retrieves the value of the `ob_size` field, and this is much faster than calling a method.

2 A C struct is a record type with named fields.

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

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

If you need to invoke a special method, it is usually better to call the related built-in function (e.g., `len`, `iter`, `str`, etc.). These built-ins call the corresponding special method, but often provide other services and—for built-in types—are faster than method calls. See, for example, Chapter 17.

In the next sections, we'll see some of the most important uses of special methods:

- Emulating numeric types
- String representation of objects
- Boolean value of an object
- Implementing collections

Emulating Numeric Types

Several special methods allow user objects to respond to operators such as +. We will cover that in more detail in Chapter 16, but here our goal is to further illustrate the use of special methods through another simple example.

We will implement a class to represent two-dimensional vectors—that is, Euclidean vectors like those used in math and physics (see Figure 1-1).

The built-in `complex` type can be used to represent two-dimensional vectors, but our class can be extended to represent *n*-dimensional vectors. We will do that in Chapter 17.

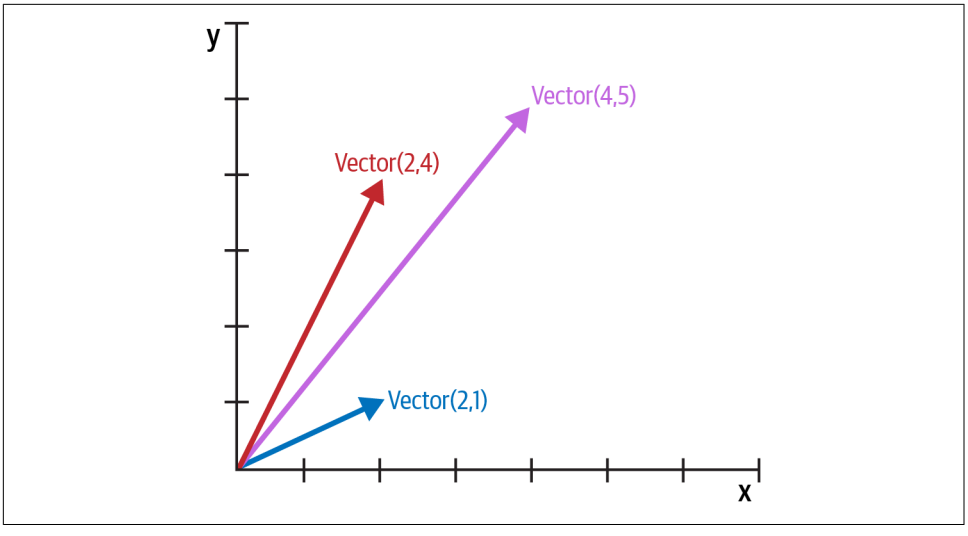


Figure 1-1. Example of two-dimensional vector addition; Vector(2, 4) + Vector(2, 1) results in Vector(4, 5).

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

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

Note how the `+` operator results in a new `Vector`, displayed in a friendly format at the console.

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

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

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

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

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

Example 1-2. A simple two-dimensional vector class

```
"""
vector2d.py: a simplistic class demonstrating some special methods

It is simplistic for didactic reasons. It lacks proper error handling,
especially in the ``__add__`` and ``__mul__`` methods.

This example is greatly expanded later in the book.

Addition::

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

Absolute value::

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

Scalar multiplication::

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

"""


import math

class Vector:

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

    def __repr__(self):
        return f'Vector({self.x!r}, {self.y!r})'

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

    def __bool__(self):
```

```
        return bool(abs(self))

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

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

We implemented five special methods in addition to the familiar `__init__`. Note that none of them is directly called within the class or in the typical usage of the class illustrated by the doctests. As mentioned before, the Python interpreter is the only frequent caller of most special methods.

Example 1-2 implements two operators: `+` and `*`, to show basic usage of `__add__` and `__mul__`. In both cases, the methods create and return a new instance of `Vector`, and do not modify either operand—`self` or `other` are merely read. This is the expected behavior of infix operators: to create new objects and not touch their operands. I will have a lot more to say about that in Chapter 16.

As implemented, Example 1-2 allows multiplying a `Vector` by a number, but not a number by a `Vector`, which violates the commutative property of scalar multiplication. We will fix that with the special method `__rmul__` in Chapter 16.

In the following sections, we discuss the other special methods in `Vector`.

String Representation

The `__repr__` special method is called by the `repr` built-in to get the string representation of the object for inspection. Without a custom `__repr__`, Python's console would display a `Vector` instance `<Vector object at 0x10e100070>`.

The interactive console and debugger call `repr` on the results of the expressions evaluated, as does the `%r` placeholder in classic formatting with the `%` operator, and the `!r` conversion field in the new format string syntax used in *f-strings* the `str.format` method.

Note that the *f-string* in our `__repr__` uses `!r` to get the standard representation of the attributes to be displayed. This is good practice, because it shows the crucial difference between `Vector(1, 2)` and `Vector('1', '2')`—the latter would not work in the context of this example, because the constructor's arguments should be numbers, not `str`.

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

In contrast, `__str__` is called by the `str()` built-in and implicitly used by the `print` function. It should return a string suitable for display to end users.

Sometimes same string returned by `__repr__` is user-friendly, and you don't need to code `__str__` because the implementation inherited from the `object` class calls `__repr__` as a fallback. Example 5-2 is one of several examples in this book with a custom `__str__`.

Programmers with prior experience in languages with a `toString` method tend to implement `__str__` and not `__repr__`. If you only implement one of these special methods in Python, choose `__repr__`.

"What is the difference between `__str__` and `__repr__` in Python?" is a Stack Overflow question with excellent contributions from Pythonistas Alex Martelli and Martijn Pieters.

Boolean Value of a Custom Type

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

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

Our implementation of `__bool__` is conceptually simple: it returns `False` if the magnitude of the vector is zero, `True` otherwise. We convert the magnitude to a Boolean using `bool(abs(self))` because `__bool__` is expected to return a Boolean. Outside of `__bool__` methods, it is rarely necessary to call `bool()` explicitly, because any object can be used in a Boolean context.

Note how the special method `__bool__` allows your objects to follow the truth value testing rules defined in the "Built-in Types" chapter of *The Python Standard Library* documentation.

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

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

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

Collection API

Figure 1-2 documents the interfaces of the essential collection types in the language. All the classes in the diagram are ABCs—*abstract base classes*. ABCs and the `collections.abc` module are covered in Chapter 13. The goal of this brief section is to give a panoramic view of Python's most important collection interfaces, showing how they are built from special methods.

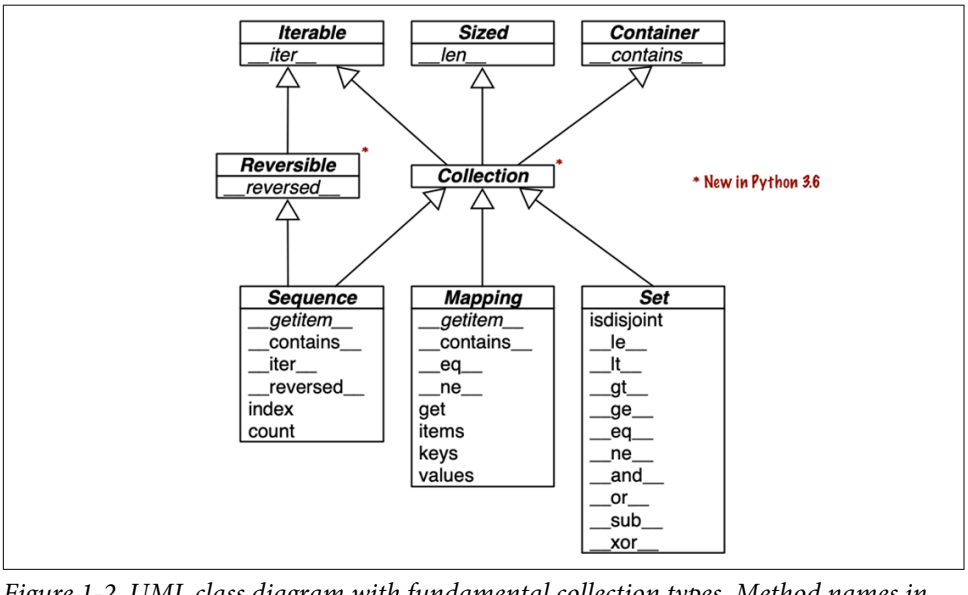


Figure 1-2. UML class diagram with fundamental collection types. Method names in italic are abstract, so they must be implemented by concrete subclasses such as `list` and `dict`. The remaining methods have concrete implementations, therefore subclasses can inherit them.

Each of the top ABCs has a single special method. The `Collection` ABC (new in Python 3.6) unifies the three essential interfaces that every collection should implement:

- `Iterable` to support `for`, unpacking, and other forms of iteration
- `Sized` to support the `len` built-in function
- `Container` to support the `in` operator

Python does not require concrete classes to actually inherit from any of these ABCs. Any class that implements `__len__` satisfies the `Sized` interface.

Three very important specializations of `Collection` are:

- `Sequence`, formalizing the interface of built-ins like `list` and `str`
- `Mapping`, implemented by `dict`, `collections.defaultdict`, etc.
- `Set`, the interface of the `set` and `frozenset` built-in types

Only `Sequence` is `Reversible`, because sequences support arbitrary ordering of their contents, while mappings and sets do not.

Since Python 3.7, the `dict` type is officially "ordered," but that only means that the key insertion order is preserved. You cannot rearrange the keys in a `dict` however you like.

All the special methods in the `Set` ABC implement infix operators. For example, `a & b` computes the intersection of sets `a` and `b`, and is implemented in the `__and__` special method.

The next two chapters will cover standard library sequences, mappings, and sets in detail.

Now let's consider the major categories of special methods defined in the Python Data Model.

Overview of Special Methods

The "Data Model" chapter of *The Python Language Reference* lists more than 80 special method names. More than half of them implement arithmetic, bitwise, and comparison operators. As an overview of what is available, see the following tables.

Table 1-1 shows special method names, excluding those used to implement infix operators or core math functions like `abs`. Most of these methods will be covered throughout the book, including the most recent additions: asynchronous special methods such as `__anext__` (added in Python 3.5), and the class customization hook, `__init_subclass__` (from Python 3.6).

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

Category	Method names
String/bytes representation	`__repr__` `__str__` `__format__` `__bytes__` `__fspath__`
Conversion to number	`__bool__` `__complex__` `__int__` `__float__` `__hash__` `__index__`
Emulating collections	`__len__` `__getitem__` `__setitem__` `__delitem__` `__contains__`
Iteration	`__iter__` `__aiter__` `__next__` `__anext__` `__reversed__`
Callable or coroutine execution	`__call__` `__await__`
Context management	`__enter__` `__exit__` `__aexit__` `__aenter__`
Instance creation and destruction	`__new__` `__init__` `__del__`
Attribute management	`__getattr__` `__getattribute__` `__setattr__` `__delattr__` `__dir__`
Attribute descriptors	`__get__` `__set__` `__delete__` `__set_name__`
Abstract base classes	`__instancecheck__` `__subclasscheck__`
Class metaprogramming	`__prepare__` `__init_subclass__` `__class_getitem__` `__mro_entries__`

Infix and numerical operators are supported by the special methods listed in Table 1-2. Here the most recent names are `__matmul__`, `__rmatmul__`, and `__imatmul__`, added in Python 3.5 to support the use of `@` as an infix operator for matrix multiplication, as we'll see in Chapter 16.

Table 1-2. Special method names and symbols for operators

Operator category	Symbols	Method names
Unary numeric	`-` `+` `abs()`	`__neg__` `__pos__` `__abs__`
Rich comparison	`<` `<=` `==` `!=` `>` `>=`	`__lt__` `__le__` `__eq__` `__ne__` `__gt__` `__ge__`
Arithmetic	`+` `-` `*` `/` `//` `%` `@` `divmod()` `round()` `**` `pow()`	`__add__` `__sub__` `__mul__` `__truediv__` `__floordiv__` `__mod__` `__matmul__` `__divmod__` `__round__` `__pow__`
Reversed arithmetic	(arithmetic operators with swapped operands)	`__radd__` `__rsub__` `__rmul__` `__rtruediv__` `__rfloordiv__` `__rmod__` `__rmatmul__` `__rdivmod__` `__rpow__`
Augmented assignment arithmetic	`+=` `-=` `*=` `/=` `//=` `%=` `@=` `**=`	`__iadd__` `__isub__` `__imul__` `__itruediv__` `__ifloordiv__` `__imod__` `__imatmul__` `__ipow__`
Bitwise	`&` `|` `^` `<<` `>>` `~`	`__and__` `__or__` `__xor__` `__lshift__` `__rshift__` `__invert__`
Reversed bitwise	(bitwise operators with swapped operands)	`__rand__` `__ror__` `__rxor__` `__rlshift__` `__rrshift__`

Operator category	Symbols	Method names
Augmented assignment bitwise	`&=` `|=` `^=` `<<=` `>>=`	`__iand__` `__ior__` `__ixor__` `__ilshift__` `__irshift__`

Python calls a reversed operator special method on the second operand when the corresponding special method on the first operand cannot be used. Augmented assignments are shortcuts combining an infix operator with variable assignment, e.g., `a += b`.

Chapter 16 explains reversed operators and augmented assignment in detail.

Why len Is Not a Method

I asked this question to core developer Raymond Hettinger in 2013, and the key to his answer was a quote from "The Zen of Python": "practicality beats purity." In "How Special Methods Are Used" on page 6, I described how `len(x)` runs very fast when `x` is an instance of a built-in type. No method is called for the built-in objects of CPython: the length is simply read from a field in a C struct. Getting the number of items in a collection is a common operation and must work efficiently for such basic and diverse types as `str`, `list`, `memoryview`, and so on.

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

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

Chapter Summary

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

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

Emulating sequences, as shown with the `FrenchDeck` example, is one of the most common uses of the special methods. For example, database libraries often return query results wrapped in sequence-like collections. Making the most of existing sequence types is the subject of Chapter 2. Implementing your own sequences will be covered in Chapter 12, when we create a multidimensional extension of the `Vector` class.

Thanks to operator overloading, Python offers a rich selection of numeric types, from the built-ins to `decimal.Decimal` and `fractions.Fraction`, all supporting infix arithmetic operators. The *NumPy* data science libraries support infix operators with matrices and tensors. Implementing operators—including reversed operators and augmented assignment—will be shown in Chapter 16 via enhancements of the `Vector` example.

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

Further Reading

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

Python in a Nutshell, 3rd ed. by Alex Martelli, Anna Ravenscroft, and Steve Holden (O'Reilly) has excellent coverage of the data model. Their description of the mechanics of attribute access is the most authoritative I've seen apart from the actual C source code of CPython. Martelli is also a prolific contributor to Stack Overflow, with more than 6,200 answers posted. See his user profile at Stack Overflow.

David Beazley has two books covering the data model in detail in the context of Python 3: *Python Essential Reference*, 4th ed. (Addison-Wesley), and *Python Cookbook*, 3rd ed. (O'Reilly), coauthored with Brian K. Jones.

The Art of the Metaobject Protocol (MIT Press) by Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow explains the concept of a metaobject protocol, of which the Python Data Model is one example.

Soapbox

Data Model or Object Model?

What the Python documentation calls the "Python Data Model," most authors would say is the "Python object model." Martelli, Ravenscroft, and Holden's *Python in a Nutshell*, 3rd ed., and David Beazley's *Python Essential Reference*, 4th ed. are the best books covering the Python Data Model, but they refer to it as the "object model." On Wikipedia, the first definition of "object model" is: "The properties of objects in general in a specific computer programming language." This is what the Python Data Model is about. In this book, I will use "data model" because the documentation favors that term when referring to the Python object model, and because it is the title of the chapter of *The Python Language Reference* most relevant to our discussions.

Muggle Methods

The Original Hacker's Dictionary defines *magic* as "yet unexplained, or too complicated to explain" or "a feature not generally publicized which allows something otherwise impossible."

The Ruby community calls their equivalent of the special methods *magic methods*. Many in the Python community adopt that term as well. I believe the special methods are the opposite of magic. Python and Ruby empower their users with a rich metaobject protocol that is fully documented, enabling muggles like you and me to emulate many of the features available to core developers who write the interpreters for those languages.

In contrast, consider Go. Some objects in that language have features that are magic, in the sense that we cannot emulate them in our own user-defined types. For example, Go arrays, strings, and maps support the use brackets for item access, as in `a[i]`. But there's no way to make the `[]` notation work with a new collection type that you define. Even worse, Go has no user-level concept of an iterable interface or an iterator object, therefore its `for/range` syntax is limited to supporting five "magic" built-in types, including arrays, strings, and maps.

Maybe in the future, the designers of Go will enhance its metaobject protocol. But currently, it is much more limited than what we have in Python or Ruby.

Metaobjects

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

A rich metaobject protocol enables extending a language to support new programming paradigms. Gregor Kiczales, the first author of the *AMOP* book, later became a pioneer in aspect-oriented programming and the initial author of AspectJ, an extension of Java implementing that paradigm. Aspect-oriented programming is much easier to implement in a dynamic language like Python, and some frameworks do it. The most important example is *zope.interface*, part of the framework on which the Plone content management system is built.

About the Author

Luciano Ramalho was a web developer before the Netscape IPO in 1995, and switched from Perl to Java to Python in 1998. He joined Thoughtworks in 2015, where he is a Principal Consultant in the São Paulo office. He has delivered keynotes, talks, and tutorials at Python events in the Americas, Europe, and Asia, and also presented at Go and Elixir conferences, focusing on language design topics. Ramalho is a fellow of the Python Software Foundation and cofounder of Garoa Hacker Clube, the first hackerspace in Brazil.

Colophon

The animal on the cover of *Fluent Python* is a Namaqua sand lizard (*Pedioplanis namaquensis*), found throughout Namibia in arid savannah and semi-desert regions.

The Namaqua sand lizard has a black body with four white stripes running down its back, brown legs with white spots, a white belly, and a long, pinkish-brown tail. It is one of the fastest of the lizards active during the day and feeds on small insects. It inhabits sparsely vegetated sand gravel flats. Female Namaqua sand lizards lay between three to five eggs in November, and these lizards spends the rest of winter dormant in burrows that they dig near the base of bushes.

The current conservation status of the Namaqua sand lizard is of "Least Concern." Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from Wood's *Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.