
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 DLR¹

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 15.
- [Figure 1-2](#) showing the use of special methods in “[Collection API](#)” on page 14, 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 struct² 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, “Using `iter` with a Callable” on page 598 in [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](#).

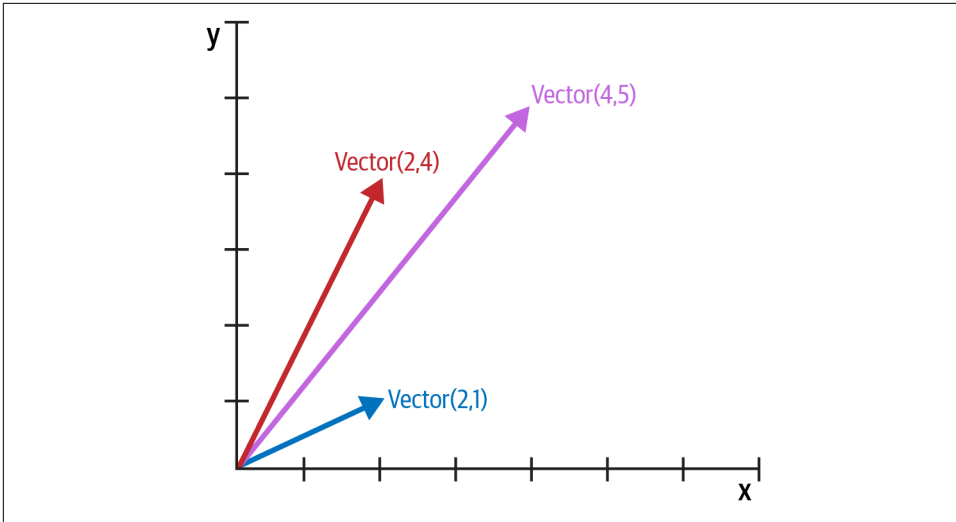


Figure 1-1. Example of two-dimensional vector addition; $\text{Vector}(2, 4) + \text{Vector}(2, 1)$ results in $\text{Vector}(4, 5)$.

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.

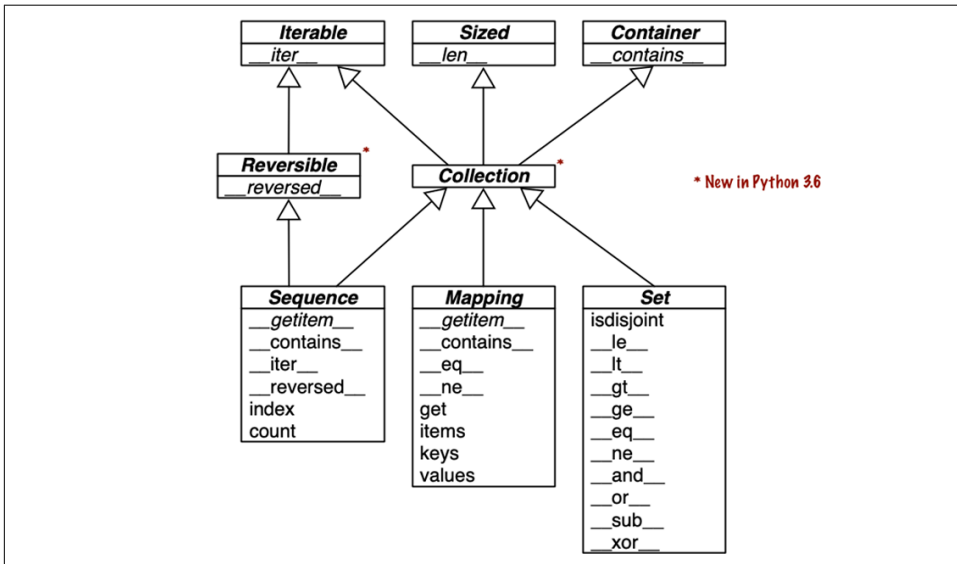


Figure 1-2. UML class diagram with fundamental collection types. Method names in *italic* are abstract, so they must be implemented by concrete subclasses such as `List` and `dict`. The remaining methods have concrete implementations, therefore subclasses can inherit them.

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

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	<code>-</code> <code>+</code> <code>abs()</code>	<code>__neg__</code> <code>__pos__</code> <code>__abs__</code>
Rich comparison	<code><</code> <code><=</code> <code>==</code> <code>!=</code> <code>></code> <code>>=</code>	<code>__lt__</code> <code>__le__</code> <code>__eq__</code> <code>__ne__</code> <code>__gt__</code> <code>__ge__</code>
Arithmetic	<code>+</code> <code>-</code> <code>*</code> <code>/</code> <code>//</code> <code>%</code> <code>@</code> <code>divmod()</code> <code>round()</code> <code>**</code> <code>pow()</code>	<code>__add__</code> <code>__sub__</code> <code>__mul__</code> <code>__truediv__</code> <code>__floordiv__</code> <code>__mod__</code> <code>__matmul__</code> <code>__div__</code> <code>__mod__</code> <code>__round__</code> <code>__pow__</code>
Reversed arithmetic	(arithmetic operators with swapped operands)	<code>__radd__</code> <code>__rsub__</code> <code>__rmul__</code> <code>__rtrue</code> <code>__div__</code> <code>__rfloordiv__</code> <code>__rmod__</code> <code>__rmat</code> <code>__mul__</code> <code>__rdivmod__</code> <code>__rpow__</code>
Augmented assignment arithmetic	<code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>//=</code> <code>%=</code> <code>@=</code> <code>**=</code>	<code>__iadd__</code> <code>__isub__</code> <code>__imul__</code> <code>__itru</code> <code>__div__</code> <code>__ifloordiv__</code> <code>__imod__</code> <code>__imat</code> <code>__mul__</code> <code>__ipow__</code>
Bitwise	<code>&</code> <code> </code> <code>^</code> <code><<</code> <code>>></code> <code>~</code>	<code>__and__</code> <code>__or__</code> <code>__xor__</code> <code>__lshift__</code> <code>__rshift__</code> <code>__invert__</code>
Reversed bitwise	(bitwise operators with swapped operands)	<code>__rand__</code> <code>__ror__</code> <code>__rxor__</code> <code>__rlshift__</code> <code>__rrshift__</code>

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



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 8, I described how `len(x)` runs very fast when `x` is an instance of a built-in type. No method is called for the built-in objects of CPython: the length is simply read from a field in a C struct. Getting the number of items in a collection is a common operation and must work efficiently for such basic and diverse types as `str`, `list`, `memoryview`, 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.

An Array of Sequences

As you may have noticed, several of the operations mentioned work equally for texts, lists and tables. Texts, lists and tables together are called ‘trains’. [...] The FOR command also works generically on trains.

—Leo Geurts, Lambert Meertens, and Steven Pembertonm, *ABC Programmer’s Handbook*¹

Before creating Python, Guido was a contributor to the ABC language—a 10-year research project to design a programming environment for beginners. ABC introduced many ideas we now consider “Pythonic”: generic operations on different types of sequences, built-in tuple and mapping types, structure by indentation, strong typing without variable declarations, and more. It’s no accident that Python is so user-friendly.

Python inherited from ABC the uniform handling of sequences. Strings, lists, byte sequences, arrays, XML elements, and database results share a rich set of common operations, including iteration, slicing, sorting, and concatenation.

Understanding the variety of sequences available in Python saves us from reinventing the wheel, and their common interface inspires us to create APIs that properly support and leverage existing and future sequence types.

Most of the discussion in this chapter applies to sequences in general, from the familiar `list` to the `str` and `bytes` types added in Python 3. Specific topics on lists, tuples, arrays, and queues are also covered here, but the specifics of Unicode strings and byte sequences appear in [Chapter 4](#). Also, the idea here is to cover sequence types that are ready to use. Creating your own sequence types is the subject of [Chapter 12](#).

¹ Leo Geurts, Lambert Meertens, and Steven Pemberton, *ABC Programmer’s Handbook*, p. 8. (Bosko Books).

These are the main topics this chapter will cover:

- List comprehensions and the basics of generator expressions
- Using tuples as records versus using tuples as immutable lists
- Sequence unpacking and sequence patterns
- Reading from slices and writing to slices
- Specialized sequence types, like arrays and queues

What's New in This Chapter

The most important update in this chapter is “[Pattern Matching with Sequences](#)” on [page 38](#). That’s the first time the new pattern matching feature of Python 3.10 appears in this second edition.

Other changes are not updates but improvements over the first edition:

- New diagram and description of the internals of sequences, contrasting containers and flat sequences
- Brief comparison of the performance and storage characteristics of `list` versus `tuple`
- Caveats of tuples with mutable elements, and how to detect them if needed

I moved coverage of named tuples to “[Classic Named Tuples](#)” on [page 169](#) in [Chapter 5](#), where they are compared to `typing.NamedTuple` and `@dataclass`.



To make room for new content and keep the page count within reason, the section “Managing Ordered Sequences with Bisect” from the first edition is now a [post](#) in the [fluentpython.com](#) companion website.

Overview of Built-In Sequences

The standard library offers a rich selection of sequence types implemented in C:

Container sequences

Can hold items of different types, including nested containers. Some examples: `list`, `tuple`, and `collections.deque`.

Flat sequences

Hold items of one simple type. Some examples: `str`, `bytes`, and `array.array`.

A *container sequence* holds references to the objects it contains, which may be of any type, while a *flat sequence* stores the value of its contents in its own memory space, not as distinct Python objects. See [Figure 2-1](#).

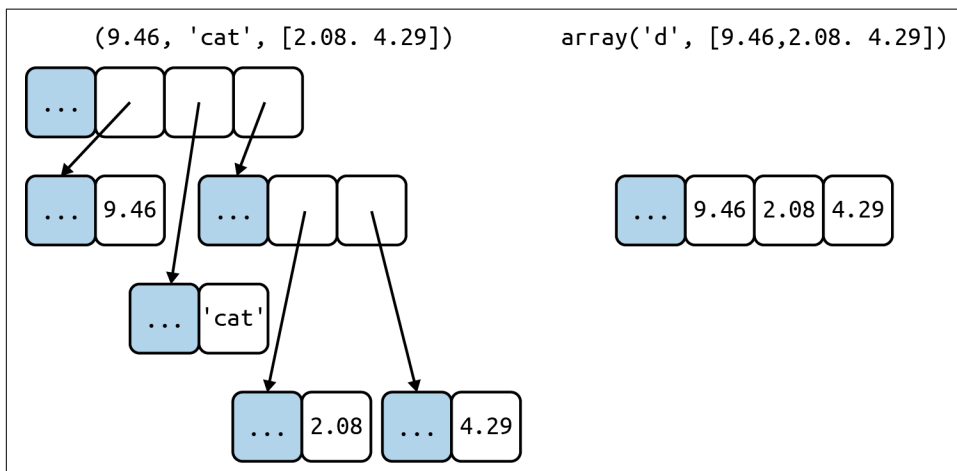


Figure 2-1. Simplified memory diagrams for a tuple and an array, each with three items. Gray cells represent the in-memory header of each Python object—not drawn to proportion. The tuple has an array of references to its items. Each item is a separate Python object, possibly holding references to other Python objects, like that two-item list. In contrast, the Python array is a single object, holding a C language array of three doubles.

Thus, flat sequences are more compact, but they are limited to holding primitive machine values like bytes, integers, and floats.



Every Python object in memory has a header with metadata. The simplest Python object, a `float`, has a value field and two metadata fields:

- `ob_refcnt`: the object's reference count
- `ob_type`: a pointer to the object's type
- `ob_fval`: a C double holding the value of the `float`

On a 64-bit Python build, each of those fields takes 8 bytes. That's why an array of floats is much more compact than a tuple of floats: the array is a single object holding the raw values of the floats, while the tuple consists of several objects—the tuple itself and each `float` object contained in it.

Another way of grouping sequence types is by mutability:

Mutable sequences

For example, `list`, `bytearray`, `array.array`, and `collections.deque`.

Immutable sequences

For example, `tuple`, `str`, and `bytes`.

Figure 2-2 helps visualize how mutable sequences inherit all methods from immutable sequences, and implement several additional methods. The built-in concrete sequence types do not actually subclass the `Sequence` and `MutableSequence` abstract base classes (ABCs), but they are *virtual subclasses* registered with those ABCs—as we’ll see in Chapter 13. Being virtual subclasses, `tuple` and `list` pass these tests:

```
>>> from collections import abc
>>> isinstance(tuple, abc.Sequence)
True
>>> isinstance(list, abc.MutableSequence)
True
```

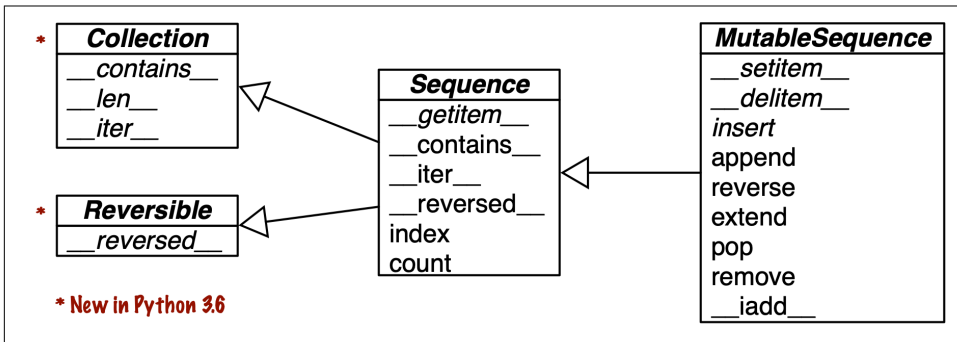


Figure 2-2. Simplified UML class diagram for some classes from `collections.abc` (superclasses are on the left; inheritance arrows point from subclasses to superclasses; names in *italic* are abstract classes and abstract methods).

Keep in mind these common traits: mutable versus immutable; container versus flat. They are helpful to extrapolate what you know about one sequence type to others.

The most fundamental sequence type is the `list`: a mutable container. I expect you are very familiar with lists, so we’ll jump right into list comprehensions, a powerful way of building lists that is sometimes underused because the syntax may look unusual at first. Mastering list comprehensions opens the door to generator expressions, which—among other uses—can produce elements to fill up sequences of any type. Both are the subject of the next section.

List Comprehensions and Generator Expressions

A quick way to build a sequence is using a list comprehension (if the target is a `list`) or a generator expression (for other kinds of sequences). If you are not using these syntactic forms on a daily basis, I bet you are missing opportunities to write code that is more readable and often faster at the same time.

If you doubt my claim that these constructs are “more readable,” read on. I’ll try to convince you.



For brevity, many Python programmers refer to list comprehensions as *listcomps*, and generator expressions as *genexps*. I will use these words as well.

List Comprehensions and Readability

Here is a test: which do you find easier to read, [Example 2-1](#) or [Example 2-2](#)?

Example 2-1. Build a list of Unicode code points from a string

```
>>> symbols = '$ç£¥€¤'
>>> codes = []
>>> for symbol in symbols:
...     codes.append(ord(symbol))
...
>>> codes
[36, 162, 163, 165, 8364, 164]
```

Example 2-2. Build a list of Unicode code points from a string, using a listcomp

```
>>> symbols = '$ç£¥€¤'
>>> codes = [ord(symbol) for symbol in symbols]
>>> codes
[36, 162, 163, 165, 8364, 164]
```

Anybody who knows a little bit of Python can read [Example 2-1](#). However, after learning about listcomps, I find [Example 2-2](#) more readable because its intent is explicit.

A for loop may be used to do lots of different things: scanning a sequence to count or pick items, computing aggregates (sums, averages), or any number of other tasks. The code in [Example 2-1](#) is building up a list. In contrast, a listcomp is more explicit. Its goal is always to build a new list.

Of course, it is possible to abuse list comprehensions to write truly incomprehensible code. I've seen Python code with listcomps used just to repeat a block of code for its side effects. If you are not doing something with the produced list, you should not use that syntax. Also, try to keep it short. If the list comprehension spans more than two lines, it is probably best to break it apart or rewrite it as a plain old `for` loop. Use your best judgment: for Python, as for English, there are no hard-and-fast rules for clear writing.



Syntax Tip

In Python code, line breaks are ignored inside pairs of `[]`, `{}`, or `()`. So you can build multiline lists, listcomps, tuples, dictionaries, etc., without using the `\` line continuation escape, which doesn't work if you accidentally type a space after it. Also, when those delimiter pairs are used to define a literal with a comma-separated series of items, a trailing comma will be ignored. So, for example, when coding a multiline list literal, it is thoughtful to put a comma after the last item, making it a little easier for the next coder to add one more item to that list, and reducing noise when reading diffs.

Local Scope Within Comprehensions and Generator Expressions

In Python 3, list comprehensions, generator expressions, and their siblings `set` and `dict` comprehensions, have a local scope to hold the variables assigned in the `for` clause.

However, variables assigned with the “Walrus operator” `:=` remain accessible after those comprehensions or expressions return—unlike local variables in a function. [PEP 572—Assignment Expressions](#) defines the scope of the target of `:=` as the enclosing function, unless there is a `global` or `nonlocal` declaration for that target.²

```
>>> x = 'ABC'
>>> codes = [ord(x) for x in x]
>>> x ❶
'ABC'
>>> codes
[65, 66, 67]
>>> codes = [last := ord(c) for c in x]
>>> last ❷
67
>>> c ❸
Traceback (most recent call last):
```

² Thanks to reader Tina Lapine for pointing this out.


```
File "<stdin>", line 1, in <module>
  NameError: name 'c' is not defined
```

- ❶ x was not clobbered: it's still bound to 'ABC'.
- ❷ last remains.
- ❸ c is gone; it existed only inside the listcomp.

List comprehensions build lists from sequences or any other iterable type by filtering and transforming items. The `filter` and `map` built-ins can be composed to do the same, but readability suffers, as we will see next.

Listcomps Versus `map` and `filter`

Listcomps do everything the `map` and `filter` functions do, without the contortions of the functionally challenged Python `lambda`. Consider [Example 2-3](#).

Example 2-3. The same list built by a listcomp and a `map/filter` composition

```
>>> symbols = '$ç£¥€¤'
>>> beyond_ascii = [ord(s) for s in symbols if ord(s) > 127]
>>> beyond_ascii
[162, 163, 165, 8364, 164]
>>> beyond_ascii = list(filter(lambda c: c > 127, map(ord, symbols)))
>>> beyond_ascii
[162, 163, 165, 8364, 164]
```

I used to believe that `map` and `filter` were faster than the equivalent listcomps, but Alex Martelli pointed out that's not the case—at least not in the preceding examples. The [02-array-seq/listcomp_speed.py](#) script in [the *Fluent Python* code repository](#) is a simple speed test comparing listcomp with `filter/map`.

I'll have more to say about `map` and `filter` in [Chapter 7](#). Now we turn to the use of listcomps to compute Cartesian products: a list containing tuples built from all items from two or more lists.

Cartesian Products

Listcomps can build lists from the Cartesian product of two or more iterables. The items that make up the Cartesian product are tuples made from items from every input iterable. The resulting list has a length equal to the lengths of the input iterables multiplied. See [Figure 2-3](#).

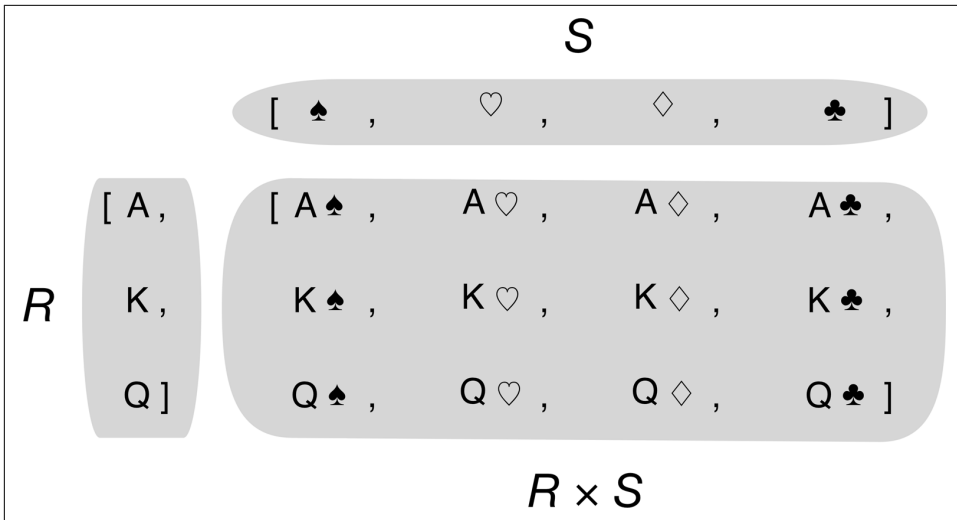


Figure 2-3. The Cartesian product of 3 card ranks and 4 suits is a sequence of 12 pairings.

For example, imagine you need to produce a list of T-shirts available in two colors and three sizes. **Example 2-4** shows how to produce that list using a listcomp. The result has six items.

Example 2-4. Cartesian product using a list comprehension

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
>>> tshirts = [(color, size) for color in colors for size in sizes] ❶
>>> tshirts
[('black', 'S'), ('black', 'M'), ('black', 'L'), ('white', 'S'),
 ('white', 'M'), ('white', 'L')]
>>> for color in colors: ❷
...     for size in sizes:
...         print((color, size))
...
('black', 'S')
('black', 'M')
('black', 'L')
('white', 'S')
('white', 'M')
('white', 'L')
>>> tshirts = [(color, size) for size in sizes ❸
...             for color in colors]
>>> tshirts
[('black', 'S'), ('white', 'S'), ('black', 'M'), ('white', 'M'),
 ('black', 'L'), ('white', 'L')]
```

- ❶ This generates a list of tuples arranged by color, then size.
- ❷ Note how the resulting list is arranged as if the for loops were nested in the same order as they appear in the listcomp.
- ❸ To get items arranged by size, then color, just rearrange the for clauses; adding a line break to the listcomp makes it easier to see how the result will be ordered.

In [Example 1-1 \(Chapter 1\)](#), I used the following expression to initialize a card deck with a list made of 52 cards from all 13 ranks of each of the 4 suits, sorted by suit, then rank:

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

Listcomps are a one-trick pony: they build lists. To generate data for other sequence types, a genexp is the way to go. The next section is a brief look at genexps in the context of building sequences that are not lists.

Generator Expressions

To initialize tuples, arrays, and other types of sequences, you could also start from a listcomp, but a genexp (generator expression) saves memory because it yields items one by one using the iterator protocol instead of building a whole list just to feed another constructor.

Genexps use the same syntax as listcomps, but are enclosed in parentheses rather than brackets.

[Example 2-5](#) shows basic usage of genexps to build a tuple and an array.

Example 2-5. Initializing a tuple and an array from a generator expression

```
>>> symbols = '§ç€¥€¤'
>>> tuple(ord(symbol) for symbol in symbols) ❶
(36, 162, 163, 165, 8364, 164)
>>> import array
>>> array.array('I', (ord(symbol) for symbol in symbols)) ❷
array('I', [36, 162, 163, 165, 8364, 164])
```

- ❶ If the generator expression is the single argument in a function call, there is no need to duplicate the enclosing parentheses.
- ❷ The array constructor takes two arguments, so the parentheses around the generator expression are mandatory. The first argument of the array constructor defines the storage type used for the numbers in the array, as we'll see in [“Arrays” on page 59](#).

Example 2-6 uses a genexp with a Cartesian product to print out a roster of T-shirts of two colors in three sizes. In contrast with **Example 2-4**, here the six-item list of T-shirts is never built in memory: the generator expression feeds the for loop producing one item at a time. If the two lists used in the Cartesian product had a thousand items each, using a generator expression would save the cost of building a list with a million items just to feed the for loop.

Example 2-6. Cartesian product in a generator expression

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
>>> for tshirt in (f'{c} {s}' for c in colors for s in sizes): ❶
...     print(tshirt)
...
black S
black M
black L
white S
white M
white L
```

- ❶ The generator expression yields items one by one; a list with all six T-shirt variations is never produced in this example.



Chapter 17 explains how generators work in detail. Here the idea was just to show the use of generator expressions to initialize sequences other than lists, or to produce output that you don't need to keep in memory.

Now we move on to the other fundamental sequence type in Python: the tuple.

Tuples Are Not Just Immutable Lists

Some introductory texts about Python present tuples as “immutable lists,” but that is short selling them. Tuples do double duty: they can be used as immutable lists and also as records with no field names. This use is sometimes overlooked, so we will start with that.

Tuples as Records

Tuples hold records: each item in the tuple holds the data for one field, and the position of the item gives its meaning.

If you think of a tuple just as an immutable list, the quantity and the order of the items may or may not be important, depending on the context. But when using a

tuple as a collection of fields, the number of items is usually fixed and their order is always important.

Example 2-7 shows tuples used as records. Note that in every expression, sorting the tuple would destroy the information because the meaning of each field is given by its position in the tuple.

Example 2-7. Tuples used as records

```
>>> lax_coordinates = (33.9425, -118.408056) ❶
>>> city, year, pop, chg, area = ('Tokyo', 2003, 32_450, 0.66, 8014) ❷
>>> traveler_ids = [('USA', '31195855'), ('BRA', 'CE342567'), ❸
...                 ('ESP', 'XDA205856')]
>>> for passport in sorted(traveler_ids): ❹
...     print('%s/%s' % passport) ❺
...
BRA/CE342567
ESP/XDA205856
USA/31195855
>>> for country, _ in traveler_ids: ❻
...     print(country)
...
USA
BRA
ESP
```

- ❶ Latitude and longitude of the Los Angeles International Airport.
- ❷ Data about Tokyo: name, year, population (thousands), population change (%), and area (km²).
- ❸ A list of tuples of the form (country_code, passport_number).
- ❹ As we iterate over the list, `passport` is bound to each tuple.
- ❺ The `%` formatting operator understands tuples and treats each item as a separate field.
- ❻ The `for` loop knows how to retrieve the items of a tuple separately—this is called “unpacking.” Here we are not interested in the second item, so we assign it to `_`, a dummy variable.



In general, using `_` as a dummy variable is just a convention. It's just a strange but valid variable name. However, in a `match/case` statement, `_` is a wildcard that matches any value but is not bound to a value. See “[Pattern Matching with Sequences](#)” on page 38. And in the Python console, the result of the preceding command is assigned to `_`—unless the result is `None`.

We often think of records as data structures with named fields. [Chapter 5](#) presents two ways of creating tuples with named fields.

But often, there's no need to go through the trouble of creating a class just to name the fields, especially if you leverage unpacking and avoid using indexes to access the fields. In [Example 2-7](#), we assigned `('Tokyo', 2003, 32_450, 0.66, 8014)` to `city`, `year`, `pop`, `chg`, `area` in a single statement. Then, the `%` operator assigned each item in the `passport` tuple to the corresponding slot in the format string in the `print` argument. Those are two examples of *tuple unpacking*.



The term *tuple unpacking* is widely used by Pythonistas, but *iterable unpacking* is gaining traction, as in the title of [PEP 3132 — Extended Iterable Unpacking](#).

“[Unpacking Sequences and Iterables](#)” on page 35 presents a lot more about unpacking not only tuples, but sequences and iterables in general.

Now let's consider the `tuple` class as an immutable variant of the `list` class.

Tuples as Immutable Lists

The Python interpreter and standard library make extensive use of tuples as immutable lists, and so should you. This brings two key benefits:

Clarity

When you see a tuple in code, you know its length will never change.

Performance

A tuple uses less memory than a `list` of the same length, and it allows Python to do some optimizations.

However, be aware that the immutability of a tuple only applies to the references contained in it. References in a tuple cannot be deleted or replaced. But if one of those references points to a mutable object, and that object is changed, then the value of the tuple changes. The next snippet illustrates this point by creating two tuples—a and b—which are initially equal. [Figure 2-4](#) represents the initial layout of the `b` tuple in memory.

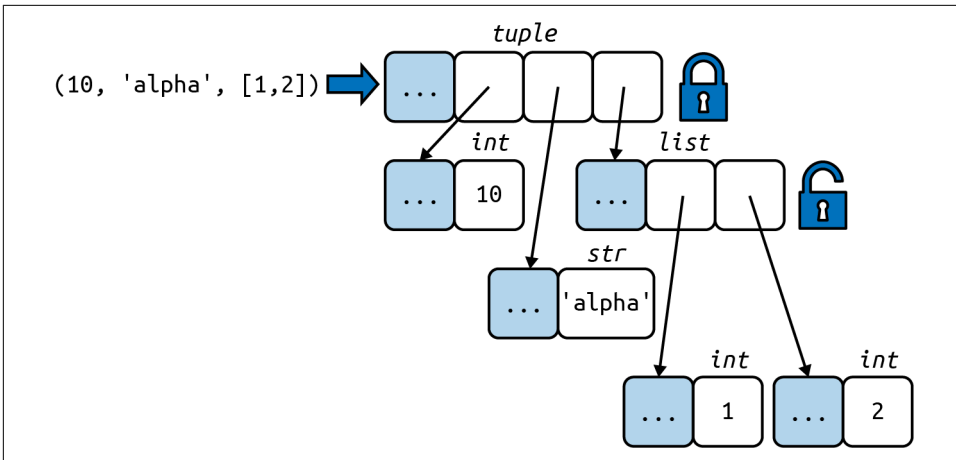


Figure 2-4. The content of the tuple itself is immutable, but that only means the references held by the tuple will always point to the same objects. However, if one of the referenced objects is mutable—like a list—its content may change.

When the last item in `b` is changed, `b` and `a` become different:

```
>>> a = (10, 'alpha', [1, 2])
>>> b = (10, 'alpha', [1, 2])
>>> a == b
True
>>> b[-1].append(99)
>>> a == b
False
>>> b
(10, 'alpha', [1, 2, 99])
```

Tuples with mutable items can be a source of bugs. As we'll see in [“What Is Hashable” on page 84](#), an object is only hashable if its value cannot ever change. An unhashable tuple cannot be inserted as a dict key, or a set element.

If you want to determine explicitly if a tuple (or any object) has a fixed value, you can use the `hash` built-in to create a fixed function like this:

```
>>> def fixed(o):
...     try:
...         hash(o)
...     except TypeError:
...         return False
...     return True
...
>>> tf = (10, 'alpha', (1, 2))
>>> tm = (10, 'alpha', [1, 2])
>>> fixed(tf)
True
```

```
>>> fixed(tm)
False
```

We explore this issue further in “The Relative Immutability of Tuples” on page 207.

Despite this caveat, tuples are widely used as immutable lists. They offer some performance advantages explained by Python core developer Raymond Hettinger in a StackOverflow answer to the question: “Are tuples more efficient than lists in Python?”. To summarize, Hettinger wrote:

- To evaluate a tuple literal, the Python compiler generates bytecode for a tuple constant in one operation; but for a list literal, the generated bytecode pushes each element as a separate constant to the data stack, and then builds the list.
- Given a tuple `t`, `tuple(t)` simply returns a reference to the same `t`. There’s no need to copy. In contrast, given a list `l`, the `list(l)` constructor must create a new copy of `l`.
- Because of its fixed length, a tuple instance is allocated the exact memory space it needs. Instances of `list`, on the other hand, are allocated with room to spare, to amortize the cost of future appends.
- The references to the items in a tuple are stored in an array in the tuple struct, while a list holds a pointer to an array of references stored elsewhere. The indirection is necessary because when a list grows beyond the space currently allocated, Python needs to reallocate the array of references to make room. The extra indirection makes CPU caches less effective.

Comparing Tuple and List Methods

When using a tuple as an immutable variation of `list`, it is good to know how similar their APIs are. As you can see in Table 2-1, `tuple` supports all `list` methods that do not involve adding or removing items, with one exception—`tuple` lacks the `__reversed__` method. However, that is just for optimization; `reversed(my_tuple)` works without it.

Table 2-1. Methods and attributes found in `list` or `tuple` (methods implemented by object are omitted for brevity)

	list	tuple	
<code>s.__add__(s2)</code>	•	•	<code>s + s2</code> —concatenation
<code>s.__iadd__(s2)</code>	•		<code>s += s2</code> —in-place concatenation
<code>s.append(e)</code>	•		Append one element after last
<code>s.clear()</code>	•		Delete all items
<code>s.__contains__(e)</code>	•	•	<code>e in s</code>
<code>s.copy()</code>	•		Shallow copy of the list

	list	tuple	
<code>s.count(e)</code>	•	•	Count occurrences of an element
<code>s.__delitem__(p)</code>	•		Remove item at position <code>p</code>
<code>s.extend(it)</code>	•		Append items from iterable <code>it</code>
<code>s.__getitem__(p)</code>	•	•	<code>s[p]</code> —get item at position
<code>s.__getnewargs__()</code>		•	Support for optimized serialization with <code>pickle</code>
<code>s.index(e)</code>	•	•	Find position of first occurrence of <code>e</code>
<code>s.insert(p, e)</code>	•		Insert element <code>e</code> before the item at position <code>p</code>
<code>s.__iter__()</code>	•	•	Get iterator
<code>s.__len__()</code>	•	•	<code>len(s)</code> —number of items
<code>s.__mul__(n)</code>	•	•	<code>s * n</code> —repeated concatenation
<code>s.__imul__(n)</code>	•		<code>s *= n</code> —in-place repeated concatenation
<code>s.__rmul__(n)</code>	•	•	<code>n * s</code> —reversed repeated concatenation ^a
<code>s.pop([p])</code>	•		Remove and return last item or item at optional position <code>p</code>
<code>s.remove(e)</code>	•		Remove first occurrence of element <code>e</code> by value
<code>s.reverse()</code>	•		Reverse the order of the items in place
<code>s.__reversed__()</code>	•		Get iterator to scan items from last to first
<code>s.__setitem__(p, e)</code>	•		<code>s[p] = e</code> —put <code>e</code> in position <code>p</code> , overwriting existing item ^b
<code>s.sort([key], [reverse])</code>	•		Sort items in place with optional keyword arguments <code>key</code> and <code>reverse</code>

^a Reversed operators are explained in [Chapter 16](#).

^b Also used to overwrite a subsequence. See [“Assigning to Slices” on page 50](#).

Now let’s switch to an important subject for idiomatic Python programming: tuple, list, and iterable unpacking.

Unpacking Sequences and Iterables

Unpacking is important because it avoids unnecessary and error-prone use of indexes to extract elements from sequences. Also, unpacking works with any iterable object as the data source—including iterators, which don’t support index notation (`[]`). The only requirement is that the iterable yields exactly one item per variable in the receiving end, unless you use a star (`*`) to capture excess items, as explained in [“Using `*` to Grab Excess Items” on page 36](#).

The most visible form of unpacking is *parallel assignment*; that is, assigning items from an iterable to a tuple of variables, as you can see in this example:

```
>>> lax_coordinates = (33.9425, -118.408056)
>>> latitude, longitude = lax_coordinates # unpacking
>>> latitude
```

```
33.9425
>>> longitude
-118.408056
```

An elegant application of unpacking is swapping the values of variables without using a temporary variable:

```
>>> b, a = a, b
```

Another example of unpacking is prefixing an argument with `*` when calling a function:

```
>>> divmod(20, 8)
(2, 4)
>>> t = (20, 8)
>>> divmod(*t)
(2, 4)
>>> quotient, remainder = divmod(*t)
>>> quotient, remainder
(2, 4)
```

The preceding code shows another use of unpacking: allowing functions to return multiple values in a way that is convenient to the caller. As another example, the `os.path.split()` function builds a tuple (`path`, `last_part`) from a filesystem path:

```
>>> import os
>>> _, filename = os.path.split('/home/luciano/.ssh/id_rsa.pub')
>>> filename
'id_rsa.pub'
```

Another way of using just some of the items when unpacking is to use the `*` syntax, as we'll see right away.

Using `*` to Grab Excess Items

Defining function parameters with `*args` to grab arbitrary excess arguments is a classic Python feature.

In Python 3, this idea was extended to apply to parallel assignment as well:

```
>>> a, b, *rest = range(5)
>>> a, b, rest
(0, 1, [2, 3, 4])
>>> a, b, *rest = range(3)
>>> a, b, rest
(0, 1, [2])
>>> a, b, *rest = range(2)
>>> a, b, rest
(0, 1, [])
```

In the context of parallel assignment, the `*` prefix can be applied to exactly one variable, but it can appear in any position:

```
>>> a, *body, c, d = range(5)
>>> a, body, c, d
(0, [1, 2], 3, 4)
>>> *head, b, c, d = range(5)
>>> head, b, c, d
([0, 1], 2, 3, 4)
```

Unpacking with * in Function Calls and Sequence Literals

PEP 448—[Additional Unpacking Generalizations](#) introduced more flexible syntax for iterable unpacking, best summarized in [“What’s New In Python 3.5”](#).

In function calls, we can use * multiple times:

```
>>> def fun(a, b, c, d, *rest):
...     return a, b, c, d, rest
...
>>> fun(*[1, 2], 3, *range(4, 7))
(1, 2, 3, 4, (5, 6))
```

The * can also be used when defining list, tuple, or set literals, as shown in these examples from [“What’s New In Python 3.5”](#):

```
>>> *range(4), 4
(0, 1, 2, 3, 4)
>>> [*range(4), 4]
[0, 1, 2, 3, 4]
>>> {*range(4), 4, *(5, 6, 7)}
{0, 1, 2, 3, 4, 5, 6, 7}
```

PEP 448 introduced similar new syntax for **, which we’ll see in [“Unpacking Mappings”](#) on page 80.

Finally, a powerful feature of tuple unpacking is that it works with nested structures.

Nested Unpacking

The target of an unpacking can use nesting, e.g., (a, b, (c, d)). Python will do the right thing if the value has the same nesting structure. [Example 2-8](#) shows nested unpacking in action.

Example 2-8. Unpacking nested tuples to access the longitude

```
metro_areas = [
    ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)), ❶
    ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
    ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
    ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
    ('São Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
]
```

```
def main():
    print(f'{"":15} | {"latitude":>9} | {"longitude":>9}')
    for name, _, _, (lat, lon) in metro_areas: ❷
        if lon <= 0: ❸
            print(f'{name:15} | {lat:9.4f} | {lon:9.4f}')

if __name__ == '__main__':
    main()
```

- ❶ Each tuple holds a record with four fields, the last of which is a coordinate pair.
- ❷ By assigning the last field to a nested tuple, we unpack the coordinates.
- ❸ The `lon <= 0`: test selects only cities in the Western hemisphere.

The output of **Example 2-8** is:

	latitude	longitude
Mexico City	19.4333	-99.1333
New York-Newark	40.8086	-74.0204
São Paulo	-23.5478	-46.6358

The target of an unpacking assignment can also be a list, but good use cases are rare. Here is the only one I know: if you have a database query that returns a single record (e.g., the SQL code has a `LIMIT 1` clause), then you can unpack and at the same time make sure there's only one result with this code:

```
>>> [record] = query_returning_single_row()
```

If the record has only one field, you can get it directly, like this:

```
>>> [[field]] = query_returning_single_row_with_single_field()
```

Both of these could be written with tuples, but don't forget the syntax quirk that single-item tuples must be written with a trailing comma. So the first target would be `(record,)` and the second `((field,),)`. In both cases you get a silent bug if you forget a comma.³

Now let's study pattern matching, which supports even more powerful ways to unpack sequences.

Pattern Matching with Sequences

The most visible new feature in Python 3.10 is pattern matching with the `match/case` statement proposed in **PEP 634—Structural Pattern Matching: Specification**.

³ Thanks to tech reviewer Leonardo Rochael for this example.



Python core developer Carol Willing wrote the excellent introduction to pattern matching in the “[Structural Pattern Matching](#)” section of “[What’s New In Python 3.10](#)”. You may want to read that quick overview. In this book, I chose to split the coverage of pattern matching over different chapters, depending on the pattern types: “[Pattern Matching with Mappings](#)” on page 81 and “[Pattern Matching Class Instances](#)” on page 192. An extended example is in “[Pattern Matching in lis.py: A Case Study](#)” on page 669.

Here is a first example of `match/case` handling sequences. Imagine you are designing a robot that accepts commands sent as sequences of words and numbers, like `BEEPER 440 3`. After splitting into parts and parsing the numbers, you’d have a message like `['BEEPER', 440, 3]`. You could use a method like this to handle such messages:

Example 2-9. Method from an imaginary Robot class

```
def handle_command(self, message):
    match message: ❶
        case ['BEEPER', frequency, times]: ❷
            self.beep(times, frequency)
        case ['NECK', angle]: ❸
            self.rotate_neck(angle)
        case ['LED', ident, intensity]: ❹
            self.leds[ident].set_brightness(ident, intensity)
        case ['LED', ident, red, green, blue]: ❺
            self.leds[ident].set_color(ident, red, green, blue)
        case _: ❻
            raise InvalidCommand(message)
```

- ❶ The expression after the `match` keyword is the *subject*. The subject is the data that Python will try to match to the patterns in each case clause.
- ❷ This pattern matches any subject that is a sequence with three items. The first item must be the string `'BEEPER'`. The second and third item can be anything, and they will be bound to the variables `frequency` and `times`, in that order.
- ❸ This matches any subject with two items, the first being `'NECK'`.
- ❹ This will match a subject with three items starting with `'LED'`. If the number of items does not match, Python proceeds to the next case.
- ❺ Another sequence pattern starting with `'LED'`, now with five items—including the `'LED'` constant.

- ⑥ This is the default case. It will match any subject that did not match a previous pattern. The `_` variable is special, as we'll soon see.

On the surface, `match/case` may look like the `switch/case` statement from the C language—but that's only half the story.⁴ One key improvement of `match` over `switch` is *destructuring*—a more advanced form of unpacking. Destructuring is a new word in the Python vocabulary, but it is commonly used in the documentation of languages that support pattern matching—like Scala and Elixir.

As a first example of destructuring, [Example 2-10](#) shows part of [Example 2-8](#) rewritten with `match/case`.

Example 2-10. Destructuring nested tuples—requires Python ≥ 3.10

```
metro_areas = [
    ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),
    ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
    ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
    ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
    ('São Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
]

def main():
    print(f'{"":15} | {"latitude":>9} | {"longitude":>9}')
    for record in metro_areas:
        match record: ①
            case [name, _, _, (lat, lon)] if lon <= 0: ②
                print(f'{name:15} | {lat:9.4f} | {lon:9.4f}')
```

- ① The subject of this `match` is `record`—i.e., each of the tuples in `metro_areas`.
- ② A case clause has two parts: a pattern and an optional guard with the `if` keyword.

In general, a sequence pattern matches the subject if:

1. The subject is a sequence *and*;
2. The subject and the pattern have the same number of items *and*;
3. Each corresponding item matches, including nested items.

⁴ In my view, a sequence of `if/elif/elif/.../else` blocks is a fine replacement for `switch/case`. It doesn't suffer from the *fallthrough* and *dangling else* problems that some language designers irrationally copied from C—decades after they were widely known as the cause of countless bugs.

For example, the pattern `[name, _, _, (lat, lon)]` in [Example 2-10](#) matches a sequence with four items, and the last item must be a two-item sequence.

Sequence patterns may be written as tuples or lists or any combination of nested tuples and lists, but it makes no difference which syntax you use: in a sequence pattern, square brackets and parentheses mean the same thing. I wrote the pattern as a list with a nested 2-tuple just to avoid repeating brackets or parentheses in [Example 2-10](#).

A sequence pattern can match instances of most actual or virtual subclasses of `collections.abc.Sequence`, with the exception of `str`, `bytes`, and `bytearray`.



Instances of `str`, `bytes`, and `bytearray` are not handled as sequences in the context of `match/case`. A `match` subject of one of those types is treated as an “atomic” value—like the integer 987 is treated as one value, not a sequence of digits. Treating those three types as sequences could cause bugs due to unintended matches. If you want to treat an object of those types as a sequence subject, convert it in the `match` clause. For example, see `tuple(phone)` in the following:

```
match tuple(phone):
    case ['1', *rest]: # North America and Caribbean
        ...
    case ['2', *rest]: # Africa and some territories
        ...
    case ['3' | '4', *rest]: # Europe
        ...
```

In the standard library, these types are compatible with sequence patterns:

<code>list</code>	<code>memoryview</code>	<code>array.array</code>
<code>tuple</code>	<code>range</code>	<code>collections.deque</code>

Unlike unpacking, patterns don’t destructure iterables that are not sequences (such as iterators).

The `_` symbol is special in patterns: it matches any single item in that position, but it is never bound to the value of the matched item. Also, the `_` is the only variable that can appear more than once in a pattern.

You can bind any part of a pattern with a variable using the `as` keyword:

```
case [name, _, _, (lat, lon) as coord]:
```

Given the subject `['Shanghai', 'CN', 24.9, (31.1, 121.3)]`, the preceding pattern will match, and set the following variables:

Variable	Set Value
name	'Shanghai'
lat	31.1
lon	121.3
coord	(31.1, 121.3)

We can make patterns more specific by adding type information. For example, the following pattern matches the same nested sequence structure as the previous example, but the first item must be an instance of `str`, and both items in the 2-tuple must be instances of `float`:

```
case [str(name), _, _, (float(lat), float(lon))]:
```



The expressions `str(name)` and `float(lat)` look like constructor calls, which we'd use to convert `name` and `lat` to `str` and `float`. But in the context of a pattern, that syntax performs a runtime type check: the preceding pattern will match a four-item sequence in which item 0 must be a `str`, and item 3 must be a pair of floats. Additionally, the `str` in item 0 will be bound to the `name` variable, and the floats in item 3 will be bound to `lat` and `lon`, respectively. So, although `str(name)` borrows the syntax of a constructor call, the semantics are completely different in the context of a pattern. Using arbitrary classes in patterns is covered in [“Pattern Matching Class Instances” on page 192](#).

On the other hand, if we want to match any subject sequence starting with a `str`, and ending with a nested sequence of two floats, we can write:

```
case [str(name), *_ , (float(lat), float(lon))]:
```

The `*_` matches any number of items, without binding them to a variable. Using `*extra` instead of `*_` would bind the items to `extra` as a list with 0 or more items.

The optional guard clause starting with `if` is evaluated only if the pattern matches, and can reference variables bound in the pattern, as in [Example 2-10](#):

```
match record:
    case [name, _, _, (lat, lon)] if lon <= 0:
        print(f'{name:15} | {lat:9.4f} | {lon:9.4f}')
```

The nested block with the `print` statement runs only if the pattern matches and the guard expression is *truthy*.



Destructuring with patterns is so expressive that sometimes a match with a single case can make code simpler. Guido van Rossum has a collection of case/match examples, including one that he titled “A very deep iterable and type match with extraction”.

Example 2-10 is not an improvement over **Example 2-8**. It’s just an example to contrast two ways of doing the same thing. The next example shows how pattern matching contributes to clear, concise, and effective code.

Pattern Matching Sequences in an Interpreter

Peter Norvig of Stanford University wrote *lis.py*: an interpreter for a subset of the Scheme dialect of the Lisp programming language in 132 lines of beautiful and readable Python code. I took Norvig’s MIT-licensed source and updated it to Python 3.10 to showcase pattern matching. In this section, we’ll compare a key part of Norvig’s code—which uses `if/elif` and unpacking—with a rewrite using `match/case`.

The two main functions of *lis.py* are `parse` and `evaluate`.⁵ The parser takes Scheme parenthesized expressions and returns Python lists. Here are two examples:

```
>>> parse('(gcd 18 45)')
['gcd', 18, 45]
>>> parse('''
... (define double
...   (lambda (n)
...     (* n 2)))
... ''')
['define', 'double', ['lambda', ['n'], ['*', 'n', 2]]]
```

The evaluator takes lists like these and executes them. The first example is calling a `gcd` function with 18 and 45 as arguments. When evaluated, it computes the greatest common divisor of the arguments: 9. The second example is defining a function named `double` with a parameter `n`. The body of the function is the expression `(* n 2)`. The result of calling a function in Scheme is the value of the last expression in its body.

Our focus here is destructuring sequences, so I will not explain the evaluator actions. See “Pattern Matching in *lis.py*: A Case Study” on page 669 to learn more about how *lis.py* works.

Example 2-11 shows Norvig’s evaluator with minor changes, abbreviated to show only the sequence patterns.

⁵ The latter is named `eval` in Norvig’s code; I renamed it to avoid confusion with Python’s `eval` built-in.

Example 2-11. Matching patterns without *match/case*

```
def evaluate(exp: Expression, env: Environment) -> Any:
    "Evaluate an expression in an environment."
    if isinstance(exp, Symbol):          # variable reference
        return env[exp]
    # ... lines omitted
    elif exp[0] == 'quote':              # (quote exp)
        (_, x) = exp
        return x
    elif exp[0] == 'if':                 # (if test conseq alt)
        (_, test, consequence, alternative) = exp
        if evaluate(test, env):
            return evaluate(consequence, env)
        else:
            return evaluate(alternative, env)
    elif exp[0] == 'lambda':            # (lambda (parm...) body...)
        (_, parms, *body) = exp
        return Procedure(parms, body, env)
    elif exp[0] == 'define':
        (_, name, value_exp) = exp
        env[name] = evaluate(value_exp, env)
    # ... more lines omitted
```

Note how each `elif` clause checks the first item of the list, and then unpacks the list, ignoring the first item. The extensive use of unpacking suggests that Norvig is a fan of pattern matching, but he wrote that code originally for Python 2 (though it now works with any Python 3).

Using `match/case` in Python ≥ 3.10 , we can refactor `evaluate` as shown in [Example 2-12](#).

Example 2-12. Pattern matching with *match/case*—requires Python ≥ 3.10

```
def evaluate(exp: Expression, env: Environment) -> Any:
    "Evaluate an expression in an environment."
    match exp:
        # ... lines omitted
        case ['quote', x]:               ❶
            return x
        case ['if', test, consequence, alternative]: ❷
            if evaluate(test, env):
                return evaluate(consequence, env)
            else:
                return evaluate(alternative, env)
        case ['lambda', [*parms], *body] if body: ❸
            return Procedure(parms, body, env)
        case ['define', Symbol() as name, value_exp]: ❹
            env[name] = evaluate(value_exp, env)
    # ... more lines omitted
```

```
case _: ❸
    raise SyntaxError(lispstr(exp))
```

- ❶ Match if subject is a two-item sequence starting with 'quote'.
- ❷ Match if subject is a four-item sequence starting with 'if'.
- ❸ Match if subject is a sequence of three or more items starting with 'lambda'. The guard ensures that body is not empty.
- ❹ Match if subject is a three-item sequence starting with 'define', followed by an instance of `Symbol`.
- ❺ It is good practice to have a catch-all case. In this example, if `exp` doesn't match any of the patterns, the expression is malformed, and I raise `SyntaxError`.

Without a catch-all, the whole `match` statement does nothing when a subject does not match any case—and this can be a silent failure.

Norvig deliberately avoided error checking in *lis.py* to keep the code easy to understand. With pattern matching, we can add more checks and still keep it readable. For example, in the 'define' pattern, the original code does not ensure that `name` is an instance of `Symbol`—that would require an `if` block, an `isinstance` call, and more code. [Example 2-12](#) is shorter and safer than [Example 2-11](#).

Alternative patterns for lambda

This is the syntax of `lambda` in Scheme, using the syntactic convention that the suffix `...` means the element may appear zero or more times:

```
(lambda (parms...) body1 body2...)
```

A simple pattern for the `lambda` case 'lambda' would be this:

```
case ['lambda', parms, *body] if body:
```

However, that matches any value in the `parms` position, including the first 'x' in this invalid subject:

```
['lambda', 'x', ['*', 'x', 2]]
```

The nested list after the `lambda` keyword in Scheme holds the names of the formal parameters for the function, and it must be a list even if it has only one element. It may also be an empty list, if the function takes no parameters—like Python's `random.random()`.

In [Example 2-12](#), I made the 'lambda' pattern safer using a nested sequence pattern:

```
case ['lambda', [*parms], *body] if body:
    return Procedure(parms, body, env)
```

In a sequence pattern, `*` can appear only once per sequence. Here we have two sequences: the outer and the inner.

Adding the characters `[*]` around `parms` made the pattern look more like the Scheme syntax it handles, and gave us an additional structural check.

Shortcut syntax for function definition

Scheme has an alternative `define` syntax to create a named function without using a nested `lambda`. This is the syntax:

```
(define (name parm...) body1 body2...)
```

The `define` keyword is followed by a list with the name of the new function and zero or more parameter names. After that list comes the function body with one or more expressions.

Adding these two lines to the `match` takes care of the implementation:

```
case ['define', [Symbol() as name, *parms], *body] if body:
    env[name] = Procedure(parms, body, env)
```

I'd place that case after the other `define` case in [Example 2-12](#). The order between the `define` cases is irrelevant in this example because no subject can match both of these patterns: the second element must be a `Symbol` in the original `define` case, but it must be a sequence starting with a `Symbol` in the `define` shortcut for function definition.

Now consider how much work we'd have adding support for this second `define` syntax without the help of pattern matching in [Example 2-11](#). The `match` statement does a lot more than the `switch` in C-like languages.

Pattern matching is an example of declarative programming: the code describes “what” you want to match, instead of “how” to match it. The shape of the code follows the shape of the data, as [Table 2-2](#) illustrates.

Table 2-2. Some Scheme syntactic forms and case patterns to handle them

Scheme syntax	Sequence pattern
(quote exp)	['quote', exp]
(if test consequent alt)	['if', test, consequent, alt]
(lambda (parms...) body1 body2...)	['lambda', [*parms], *body] if body
(define name exp)	['define', Symbol() as name, exp]
(define (name parms...) body1 body2...)	['define', [Symbol() as name, *parms], *body] if body

I hope this refactoring of Norvig’s `evaluate` with pattern matching convinced you that `match/case` can make your code more readable and safer.



We’ll see more of *lis.py* in “[Pattern Matching in lis.py: A Case Study](#)” on page 669, when we’ll review the complete `match/case` example in `evaluate`. If you want to learn more about Norvig’s *lis.py*, read his wonderful post “[\(How to Write a \(Lisp\) Interpreter \(in Python\)\)](#)”.

This concludes our first tour of unpacking, destructuring, and pattern matching with sequences. We’ll cover other types of patterns in later chapters.

Every Python programmer knows that sequences can be sliced using the `s[a:b]` syntax. We now turn to some less well-known facts about slicing.

Slicing

A common feature of `list`, `tuple`, `str`, and all sequence types in Python is the support of slicing operations, which are more powerful than most people realize.

In this section, we describe the *use* of these advanced forms of slicing. Their implementation in a user-defined class will be covered in [Chapter 12](#), in keeping with our philosophy of covering ready-to-use classes in this part of the book, and creating new classes in [Part III](#).

Why Slices and Ranges Exclude the Last Item

The Pythonic convention of excluding the last item in slices and ranges works well with the zero-based indexing used in Python, C, and many other languages. Some convenient features of the convention are:

- It’s easy to see the length of a slice or range when only the stop position is given: `range(3)` and `my_list[:3]` both produce three items.
- It’s easy to compute the length of a slice or range when start and stop are given: just subtract `stop - start`.
- It’s easy to split a sequence in two parts at any index `x`, without overlapping: simply get `my_list[:x]` and `my_list[x:]`. For example:

```
>>> l = [10, 20, 30, 40, 50, 60]
>>> l[:2] # split at 2
[10, 20]
>>> l[2:]
[30, 40, 50, 60]
>>> l[:3] # split at 3
[10, 20, 30]
```

```
>>> l[3:]
[40, 50, 60]
```

The best arguments for this convention were written by the Dutch computer scientist Edsger W. Dijkstra (see the last reference in “[Further Reading](#)” on page 71).

Now let’s take a close look at how Python interprets slice notation.

Slice Objects

This is no secret, but worth repeating just in case: `s[a:b:c]` can be used to specify a stride or step `c`, causing the resulting slice to skip items. The stride can also be negative, returning items in reverse. Three examples make this clear:

```
>>> s = 'bicycle'
>>> s[::3]
'bye'
>>> s[::-1]
'elcycib'
>>> s[::-2]
'eccb'
```

Another example was shown in [Chapter 1](#) when we used `deck[12::13]` to get all the aces in the unshuffled deck:

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

The notation `a:b:c` is only valid within `[]` when used as the indexing or subscript operator, and it produces a slice object: `slice(a, b, c)`. As we will see in “[How Slicing Works](#)” on page 404, to evaluate the expression `seq[start:stop:step]`, Python calls `seq.__getitem__(slice(start, stop, step))`. Even if you are not implementing your own sequence types, knowing about slice objects is useful because it lets you assign names to slices, just like spreadsheets allow naming of cell ranges.

Suppose you need to parse flat-file data like the invoice shown in [Example 2-13](#). Instead of filling your code with hardcoded slices, you can name them. See how readable this makes the for loop at the end of the example.

Example 2-13. Line items from a flat-file invoice

```
>>> invoice = """
... 0.....6.....40.....52...55.....
... 1909 Pimoroni PiBrella                $17.50  3   $52.50
... 1489 6mm Tactile Switch x20           $4.95   2   $9.90
... 1510 Panavise Jr. - PV-201            $28.00  1   $28.00
... 1601 PiTFT Mini Kit 320x240          $34.95  1   $34.95
... """
```

```

>>> SKU = slice(0, 6)
>>> DESCRIPTION = slice(6, 40)
>>> UNIT_PRICE = slice(40, 52)
>>> QUANTITY = slice(52, 55)
>>> ITEM_TOTAL = slice(55, None)
>>> line_items = invoice.split('\n')[2:]
>>> for item in line_items:
...     print(item[UNIT_PRICE], item[DESCRIPTION])
...
$17.50   Pimoroni PiBrella
$4.95    6mm Tactile Switch x20
$28.00   Panavise Jr. - PV-201
$34.95   PiTFT Mini Kit 320x240

```

We'll come back to slice objects when we discuss creating your own collections in [“Vector Take #2: A Sliceable Sequence” on page 403](#). Meanwhile, from a user perspective, slicing includes additional features such as multidimensional slices and ellipsis (...) notation. Read on.

Multidimensional Slicing and Ellipsis

The `[]` operator can also take multiple indexes or slices separated by commas. The `__getitem__` and `__setitem__` special methods that handle the `[]` operator simply receive the indices in `a[i, j]` as a tuple. In other words, to evaluate `a[i, j]`, Python calls `a.__getitem__((i, j))`.

This is used, for instance, in the external NumPy package, where items of a two-dimensional `numpy.ndarray` can be fetched using the syntax `a[i, j]` and a two-dimensional slice obtained with an expression like `a[m:n, k:l]`. [Example 2-22](#) later in this chapter shows the use of this notation.

Except for `memoryview`, the built-in sequence types in Python are one-dimensional, so they support only one index or slice, and not a tuple of them.⁶

The ellipsis—written with three full stops (...) and not ... (Unicode U+2026)—is recognized as a token by the Python parser. It is an alias to the `Ellipsis` object, the single instance of the `ellipsis` class.⁷ As such, it can be passed as an argument to functions and as part of a slice specification, as in `f(a, ..., z)` or `a[i:...]`. NumPy uses ... as a shortcut when slicing arrays of many dimensions; for example,

⁶ In [“Memory Views” on page 62](#) we show that specially constructed memory views can have more than one dimension.

⁷ No, I did not get this backwards: the `ellipsis` class name is really all lowercase, and the instance is a built-in named `Ellipsis`, just like `bool` is lowercase but its instances are `True` and `False`.

if `x` is a four-dimensional array, `x[i, ...]` is a shortcut for `x[i, :, :, :]`. See “NumPy quickstart” to learn more about this.

At the time of this writing, I am unaware of uses of Ellipsis or multidimensional indexes and slices in the Python standard library. If you spot one, let me know. These syntactic features exist to support user-defined types and extensions such as NumPy.

Slices are not just useful to extract information from sequences; they can also be used to change mutable sequences in place—that is, without rebuilding them from scratch.

Assigning to Slices

Mutable sequences can be grafted, excised, and otherwise modified in place using slice notation on the lefthand side of an assignment statement or as the target of a `del` statement. The next few examples give an idea of the power of this notation:

```
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[2:5] = [20, 30]
>>> l
[0, 1, 20, 30, 5, 6, 7, 8, 9]
>>> del l[5:7]
>>> l
[0, 1, 20, 30, 5, 8, 9]
>>> l[3::2] = [11, 22]
>>> l
[0, 1, 20, 11, 5, 22, 9]
>>> l[2:5] = 100 ❶
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only assign an iterable
>>> l[2:5] = [100]
>>> l
[0, 1, 100, 22, 9]
```

- ❶ When the target of the assignment is a slice, the righthand side must be an iterable object, even if it has just one item.

Every coder knows that concatenation is a common operation with sequences. Introductory Python tutorials explain the use of `+` and `*` for that purpose, but there are some subtle details on how they work, which we cover next.

Using `+` and `*` with Sequences

Python programmers expect that sequences support `+` and `*`. Usually both operands of `+` must be of the same sequence type, and neither of them is modified, but a new sequence of that same type is created as result of the concatenation.

To concatenate multiple copies of the same sequence, multiply it by an integer. Again, a new sequence is created:

```
>>> l = [1, 2, 3]
>>> l * 5
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 5 * 'abcd'
'abcdabcdabcdabcd'
```

Both + and * always create a new object, and never change their operands.



Beware of expressions like `a * n` when `a` is a sequence containing mutable items, because the result may surprise you. For example, trying to initialize a list of lists as `my_list = [[]] * 3` will result in a list with three references to the same inner list, which is probably not what you want.

The next section covers the pitfalls of trying to use * to initialize a list of lists.

Building Lists of Lists

Sometimes we need to initialize a list with a certain number of nested lists—for example, to distribute students in a list of teams or to represent squares on a game board. The best way of doing so is with a list comprehension, as in [Example 2-14](#).

Example 2-14. A list with three lists of length 3 can represent a tic-tac-toe board

```
>>> board = [['_' * 3 for i in range(3)] ❶
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[1][2] = 'X' ❷
>>> board
[['_', '_', '_'], ['_', '_', 'X'], ['_', '_', '_']]
```

❶ Create a list of three lists of three items each. Inspect the structure.

❷ Place a mark in row 1, column 2, and check the result.

A tempting, but wrong, shortcut is doing it like [Example 2-15](#).

Example 2-15. A list with three references to the same list is useless

```
>>> weird_board = [['_' * 3] * 3] ❶
>>> weird_board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> weird_board[1][2] = 'O' ❷
```

```
>>> weird_board
[['_', '_', '0'], ['_', '_', '0'], ['_', '_', '0']]
```

- ❶ The outer list is made of three references to the same inner list. While it is unchanged, all seems right.
- ❷ Placing a mark in row 1, column 2, reveals that all rows are aliases referring to the same object.

The problem with [Example 2-15](#) is that, in essence, it behaves like this code:

```
row = ['_'] * 3
board = []
for i in range(3):
    board.append(row) ❶
```

- ❶ The same row is appended three times to board.

On the other hand, the list comprehension from [Example 2-14](#) is equivalent to this code:

```
>>> board = []
>>> for i in range(3):
...     row = ['_'] * 3 ❶
...     board.append(row)
...
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[2][0] = 'X'
>>> board ❷
[['_', '_', '_'], ['_', '_', '_'], ['X', '_', '_']]
```

- ❶ Each iteration builds a new row and appends it to board.
- ❷ Only row 2 is changed, as expected.



If either the problem or the solution in this section is not clear to you, relax. [Chapter 6](#) was written to clarify the mechanics and pitfalls of references and mutable objects.

So far we have discussed the use of the plain + and * operators with sequences, but there are also the += and *= operators, which produce very different results, depending on the mutability of the target sequence. The following section explains how that works.

Augmented Assignment with Sequences

The augmented assignment operators `+=` and `*=` behave quite differently, depending on the first operand. To simplify the discussion, we will focus on augmented addition first (`+=`), but the concepts also apply to `*=` and to other augmented assignment operators.

The special method that makes `+=` work is `__iadd__` (for “in-place addition”).

However, if `__iadd__` is not implemented, Python falls back to calling `__add__`. Consider this simple expression:

```
>>> a += b
```

If `a` implements `__iadd__`, that will be called. In the case of mutable sequences (e.g., `list`, `bytearray`, `array.array`), `a` will be changed in place (i.e., the effect will be similar to `a.extend(b)`). However, when `a` does not implement `__iadd__`, the expression `a += b` has the same effect as `a = a + b`: the expression `a + b` is evaluated first, producing a new object, which is then bound to `a`. In other words, the identity of the object bound to `a` may or may not change, depending on the availability of `__iadd__`.

In general, for mutable sequences, it is a good bet that `__iadd__` is implemented and that `+=` happens in place. For immutable sequences, clearly there is no way for that to happen.

What I just wrote about `+=` also applies to `*=`, which is implemented via `__imul__`. The `__iadd__` and `__imul__` special methods are discussed in [Chapter 16](#). Here is a demonstration of `*=` with a mutable sequence and then an immutable one:

```
>>> l = [1, 2, 3]
>>> id(l)
4311953800 ❶
>>> l *= 2
>>> l
[1, 2, 3, 1, 2, 3]
>>> id(l)
4311953800 ❷
>>> t = (1, 2, 3)
>>> id(t)
4312681568 ❸
>>> t *= 2
>>> id(t)
4301348296 ❹
```

❶ ID of the initial list.

❷ After multiplication, the list is the same object, with new items appended.

- ③ ID of the initial tuple.
- ④ After multiplication, a new tuple was created.

Repeated concatenation of immutable sequences is inefficient, because instead of just appending new items, the interpreter has to copy the whole target sequence to create a new one with the new items concatenated.⁸

We’ve seen common use cases for `+=`. The next section shows an intriguing corner case that highlights what “immutable” really means in the context of tuples.

A `+=` Assignment Puzzler

Try to answer without using the console: what is the result of evaluating the two expressions in [Example 2-16](#)?⁹

Example 2-16. A riddle

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
```

What happens next? Choose the best answer:

- A. `t` becomes `(1, 2, [30, 40, 50, 60])`.
- B. `TypeError` is raised with the message `'tuple' object does not support item assignment`.
- C. Neither.
- D. Both A and B.

When I saw this, I was pretty sure the answer was B, but it’s actually D, “Both A and B”! [Example 2-17](#) is the actual output from a Python 3.9 console.¹⁰

⁸ `str` is an exception to this description. Because string building with `+=` in loops is so common in real codebases, CPython is optimized for this use case. Instances of `str` are allocated in memory with extra room, so that concatenation does not require copying the whole string every time.

⁹ Thanks to Leonardo Rocha and Cesar Kawakami for sharing this riddle at the 2013 PythonBrasil Conference.

¹⁰ Readers suggested that the operation in the example can be done with `t[2].extend([50,60])`, without errors. I am aware of that, but my intent is to show the strange behavior of the `+=` operator in this case.

Example 2-17. The unexpected result: item t2 is changed and an exception is raised

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, [30, 40, 50, 60])
```

Online Python Tutor is an awesome online tool to visualize how Python works in detail. Figure 2-5 is a composite of two screenshots showing the initial and final states of the tuple `t` from Example 2-17.

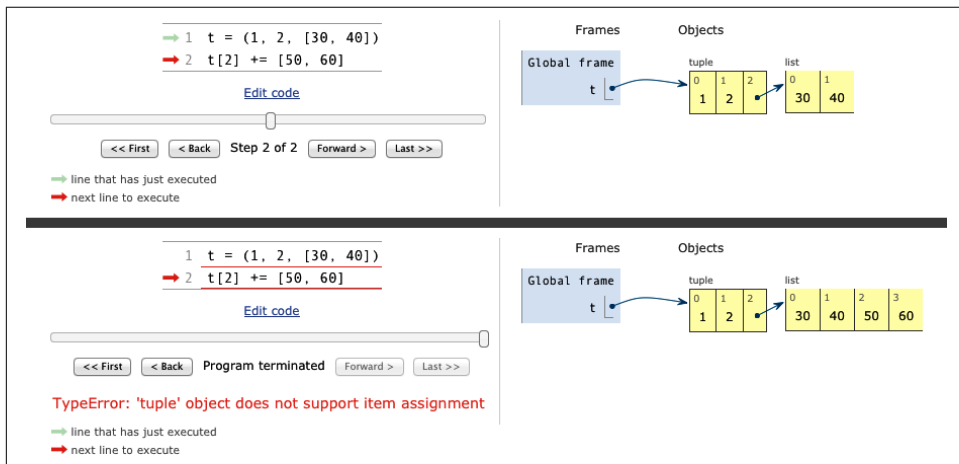


Figure 2-5. Initial and final state of the tuple assignment puzzler (diagram generated by Online Python Tutor).

If you look at the bytecode Python generates for the expression `s[a] += b` (Example 2-18), it becomes clear how that happens.

Example 2-18. Bytecode for the expression `s[a] += b`

```
>>> dis.dis('s[a] += b')
1          0 LOAD_NAME          0 (s)
          3 LOAD_NAME          1 (a)
          6 DUP_TOP_TWO
          7 BINARY_SUBSCR
          8 LOAD_NAME          2 (b)
         11 INPLACE_ADD
         12 ROT_THREE
         13 STORE_SUBSCR
         14 LOAD_CONST        0 (None)
         17 RETURN_VALUE
```

- ❶ Put the value of `s[a]` on TOS (Top Of Stack).
- ❷ Perform `TOS += b`. This succeeds if TOS refers to a mutable object (it's a list, in [Example 2-17](#)).
- ❸ Assign `s[a] = TOS`. This fails if `s` is immutable (the `t` tuple in [Example 2-17](#)).

This example is quite a corner case—in 20 years using Python, I have never seen this strange behavior actually bite somebody.

I take three lessons from this:

- Avoid putting mutable items in tuples.
- Augmented assignment is not an atomic operation—we just saw it throwing an exception after doing part of its job.
- Inspecting Python bytecode is not too difficult, and can be helpful to see what is going on under the hood.

After witnessing the subtleties of using `+` and `*` for concatenation, we can change the subject to another essential operation with sequences: sorting.

list.sort Versus the sorted Built-In

The `list.sort` method sorts a list in place—that is, without making a copy. It returns `None` to remind us that it changes the receiver¹¹ and does not create a new list. This is an important Python API convention: functions or methods that change an object in place should return `None` to make it clear to the caller that the receiver was changed, and no new object was created. Similar behavior can be seen, for example, in the `random.shuffle(s)` function, which shuffles the mutable sequence `s` in place, and returns `None`.



The convention of returning `None` to signal in-place changes has a drawback: we cannot cascade calls to those methods. In contrast, methods that return new objects (e.g., all `str` methods) can be cascaded in the fluent interface style. See Wikipedia's "[Fluent interface](#)" entry for further description of this topic.

In contrast, the built-in function `sorted` creates a new list and returns it. It accepts any iterable object as an argument, including immutable sequences and generators

¹¹ Receiver is the target of a method call, the object bound to `self` in the method body.

(see [Chapter 17](#)). Regardless of the type of iterable given to `sorted`, it always returns a newly created list.

Both `list.sort` and `sorted` take two optional, keyword-only arguments:

`reverse`

If `True`, the items are returned in descending order (i.e., by reversing the comparison of the items). The default is `False`.

`key`

A one-argument function that will be applied to each item to produce its sorting key. For example, when sorting a list of strings, `key=str.lower` can be used to perform a case-insensitive sort, and `key=len` will sort the strings by character length. The default is the identity function (i.e., the items themselves are compared).



You can also use the optional keyword parameter `key` with the `min()` and `max()` built-ins and with other functions from the standard library (e.g., `itertools.groupby()` and `heapq.nlargest()`).

Here are a few examples to clarify the use of these functions and keyword arguments. The examples also demonstrate that Python’s sorting algorithm is stable (i.e., it preserves the relative ordering of items that compare equally):¹²

```
>>> fruits = ['grape', 'raspberry', 'apple', 'banana']
>>> sorted(fruits)
['apple', 'banana', 'grape', 'raspberry'] ❶
>>> fruits
['grape', 'raspberry', 'apple', 'banana'] ❷
>>> sorted(fruits, reverse=True)
['raspberry', 'grape', 'banana', 'apple'] ❸
>>> sorted(fruits, key=len)
['grape', 'apple', 'banana', 'raspberry'] ❹
>>> sorted(fruits, key=len, reverse=True)
['raspberry', 'banana', 'grape', 'apple'] ❺
>>> fruits
['grape', 'raspberry', 'apple', 'banana'] ❻
>>> fruits.sort()
>>> fruits
['apple', 'banana', 'grape', 'raspberry'] ❼
```

¹² Python’s main sorting algorithm is named Timsort after its creator, Tim Peters. For a bit of Timsort trivia, see the [“Soapbox”](#) on [page 73](#).

- ❶ This produces a new list of strings sorted alphabetically.¹³
- ❷ Inspecting the original list, we see it is unchanged.
- ❸ This is the previous “alphabetical” ordering, reversed.
- ❹ A new list of strings, now sorted by length. Because the sorting algorithm is stable, “grape” and “apple,” both of length 5, are in the original order.
- ❺ These are the strings sorted by length in descending order. It is not the reverse of the previous result because the sorting is stable, so again “grape” appears before “apple.”
- ❻ So far, the ordering of the original `fruits` list has not changed.
- ❼ This sorts the list in place, and returns `None` (which the console omits).
- ❽ Now `fruits` is sorted.



By default, Python sorts strings lexicographically by character code. That means ASCII uppercase letters will come before lowercase letters, and non-ASCII characters are unlikely to be sorted in a sensible way. “[Sorting Unicode Text](#)” on [page 148](#) covers proper ways of sorting text as humans would expect.

Once your sequences are sorted, they can be very efficiently searched. A binary search algorithm is already provided in the `bisect` module of the Python standard library. That module also includes the `bisect.insort` function, which you can use to make sure that your sorted sequences stay sorted. You’ll find an illustrated introduction to the `bisect` module in the “[Managing Ordered Sequences with Bisect](#)” post in the *fluentpython.com* companion website.

Much of what we have seen so far in this chapter applies to sequences in general, not just lists or tuples. Python programmers sometimes overuse the `list` type because it is so handy—I know I’ve done it. For example, if you are processing large lists of numbers, you should consider using arrays instead. The remainder of the chapter is devoted to alternatives to lists and tuples.

¹³ The words in this example are sorted alphabetically because they are 100% made of lowercase ASCII characters. See the warning after the example.

When a List Is Not the Answer

The `list` type is flexible and easy to use, but depending on specific requirements, there are better options. For example, an array saves a lot of memory when you need to handle millions of floating-point values. On the other hand, if you are constantly adding and removing items from opposite ends of a list, it's good to know that a deque (double-ended queue) is a more efficient FIFO¹⁴ data structure.



If your code frequently checks whether an item is present in a collection (e.g., `item in my_collection`), consider using a set for `my_collection`, especially if it holds a large number of items. Sets are optimized for fast membership checking. They are also iterable, but they are not sequences because the ordering of set items is unspecified. We cover them in [Chapter 3](#).

For the remainder of this chapter, we discuss mutable sequence types that can replace lists in many cases, starting with arrays.

Arrays

If a list only contains numbers, an `array.array` is a more efficient replacement. Arrays support all mutable sequence operations (including `.pop`, `.insert`, and `.extend`), as well as additional methods for fast loading and saving, such as `.frombytes` and `.tofile`.

A Python array is as lean as a C array. As shown in [Figure 2-1](#), an array of float values does not hold full-fledged float instances, but only the packed bytes representing their machine values—similar to an array of double in the C language. When creating an array, you provide a `typecode`, a letter to determine the underlying C type used to store each item in the array. For example, `b` is the typecode for what C calls a signed char, an integer ranging from -128 to 127. If you create an `array('b')`, then each item will be stored in a single byte and interpreted as an integer. For large sequences of numbers, this saves a lot of memory. And Python will not let you put any number that does not match the type for the array.

[Example 2-19](#) shows creating, saving, and loading an array of 10 million floating-point random numbers.

¹⁴ First in, first out—the default behavior of queues.

Example 2-19. Creating, saving, and loading a large array of floats

```
>>> from array import array ❶
>>> from random import random
>>> floats = array('d', (random() for i in range(10**7))) ❷
>>> floats[-1] ❸
0.07802343889111107
>>> fp = open('floats.bin', 'wb')
>>> floats.tofile(fp) ❹
>>> fp.close()
>>> floats2 = array('d') ❺
>>> fp = open('floats.bin', 'rb')
>>> floats2.fromfile(fp, 10**7) ❻
>>> fp.close()
>>> floats2[-1] ❼
0.07802343889111107
>>> floats2 == floats ❽
True
```

- ❶ Import the array type.
- ❷ Create an array of double-precision floats (typecode 'd') from any iterable object—in this case, a generator expression.
- ❸ Inspect the last number in the array.
- ❹ Save the array to a binary file.
- ❺ Create an empty array of doubles.
- ❻ Read 10 million numbers from the binary file.
- ❼ Inspect the last number in the array.
- ❽ Verify that the contents of the arrays match.

As you can see, `array.tofile` and `array.fromfile` are easy to use. If you try the example, you'll notice they are also very fast. A quick experiment shows that it takes about 0.1 seconds for `array.fromfile` to load 10 million double-precision floats from a binary file created with `array.tofile`. That is nearly 60 times faster than reading the numbers from a text file, which also involves parsing each line with the `float` built-in. Saving with `array.tofile` is about seven times faster than writing one float per line in a text file. In addition, the size of the binary file with 10 million doubles is 80,000,000 bytes (8 bytes per double, zero overhead), while the text file has 181,515,739 bytes for the same data.

For the specific case of numeric arrays representing binary data, such as raster images, Python has the `bytes` and `bytearray` types discussed in [Chapter 4](#).

We wrap up this section on arrays with [Table 2-3](#), comparing the features of `list` and `array.array`.

Table 2-3. Methods and attributes found in `list` or `array` (deprecated array methods and those also implemented by object are omitted for brevity)

	list	array
<code>s.__add__(s2)</code>	•	• <code>s + s2</code> —concatenation
<code>s.__iadd__(s2)</code>	•	• <code>s += s2</code> —in-place concatenation
<code>s.append(e)</code>	•	• Append one element after last
<code>s.byteswap()</code>		• Swap bytes of all items in array for endianness conversion
<code>s.clear()</code>	•	• Delete all items
<code>s.__contains__(e)</code>	•	• <code>e in s</code>
<code>s.copy()</code>	•	• Shallow copy of the list
<code>s.__copy__()</code>		• Support for <code>copy.copy</code>
<code>s.count(e)</code>	•	• Count occurrences of an element
<code>s.__deepcopy__()</code>		• Optimized support for <code>copy.deepcopy</code>
<code>s.__delitem__(p)</code>	•	• Remove item at position <code>p</code>
<code>s.extend(it)</code>	•	• Append items from iterable <code>it</code>
<code>s.frombytes(b)</code>		• Append items from byte sequence interpreted as packed machine values
<code>s.fromfile(f, n)</code>		• Append <code>n</code> items from binary file <code>f</code> interpreted as packed machine values
<code>s.fromlist(l)</code>		• Append items from list; if one causes <code>TypeError</code> , none are appended
<code>s.__getitem__(p)</code>	•	• <code>s[p]</code> —get item or slice at position
<code>s.index(e)</code>	•	• Find position of first occurrence of <code>e</code>
<code>s.insert(p, e)</code>	•	• Insert element <code>e</code> before the item at position <code>p</code>
<code>s.itemsize</code>		• Length in bytes of each array item
<code>s.__iter__()</code>	•	• Get iterator
<code>s.__len__()</code>	•	• <code>len(s)</code> —number of items
<code>s.__mul__(n)</code>	•	• <code>s * n</code> —repeated concatenation
<code>s.__imul__(n)</code>	•	• <code>s *= n</code> —in-place repeated concatenation
<code>s.__rmul__(n)</code>	•	• <code>n * s</code> —reversed repeated concatenation ^a
<code>s.pop([p])</code>	•	• Remove and return item at position <code>p</code> (default: last)
<code>s.remove(e)</code>	•	• Remove first occurrence of element <code>e</code> by value
<code>s.reverse()</code>	•	• Reverse the order of the items in place
<code>s.__reversed__()</code>	•	• Get iterator to scan items from last to first
<code>s.__setitem__(p, e)</code>	•	• <code>s[p] = e</code> —put <code>e</code> in position <code>p</code> , overwriting existing item or slice

	list	array
<code>s.sort([key], [reverse])</code>	•	Sort items in place with optional keyword arguments <code>key</code> and <code>reverse</code>
<code>s.tobytes()</code>	•	Return items as packed machine values in a bytes object
<code>s.tofile(f)</code>	•	Save items as packed machine values to binary file <code>f</code>
<code>s.tolist()</code>	•	Return items as numeric objects in a list
<code>s.typecode</code>	•	One-character string identifying the C type of the items

^a Reversed operators are explained in [Chapter 16](#).



As of Python 3.10, the array type does not have an in-place sort method like `list.sort()`. If you need to sort an array, use the built-in `sorted` function to rebuild the array:

```
a = array.array(a.typecode, sorted(a))
```

To keep a sorted array sorted while adding items to it, use the `bisect.insort` function.

If you do a lot of work with arrays and don't know about `memoryview`, you're missing out. See the next topic.

Memory Views

The built-in `memoryview` class is a shared-memory sequence type that lets you handle slices of arrays without copying bytes. It was inspired by the NumPy library (which we'll discuss shortly in [“NumPy” on page 64](#)). Travis Oliphant, lead author of NumPy, answers the question, [“When should a memoryview be used?”](#) like this:

A memoryview is essentially a generalized NumPy array structure in Python itself (without the math). It allows you to share memory between data-structures (things like PIL images, SQLite databases, NumPy arrays, etc.) without first copying. This is very important for large data sets.

Using notation similar to the array module, the `memoryview.cast` method lets you change the way multiple bytes are read or written as units without moving bits around. `memoryview.cast` returns yet another `memoryview` object, always sharing the same memory.

[Example 2-20](#) shows how to create alternate views on the same array of 6 bytes, to operate on it as a 2×3 matrix or a 3×2 matrix.

Example 2-20. Handling 6 bytes of memory as 1×6, 2×3, and 3×2 views

```
>>> from array import array
>>> octets = array('B', range(6)) ❶
```

```

>>> m1 = memoryview(octets) ❷
>>> m1.tolist()
[0, 1, 2, 3, 4, 5]
>>> m2 = m1.cast('B', [2, 3]) ❸
>>> m2.tolist()
[[0, 1, 2], [3, 4, 5]]
>>> m3 = m1.cast('B', [3, 2]) ❹
>>> m3.tolist()
[[0, 1], [2, 3], [4, 5]]
>>> m2[1,1] = 22 ❺
>>> m3[1,1] = 33 ❻
>>> octets ❼
array('B', [0, 1, 2, 33, 22, 5])

```

- ❶ Build array of 6 bytes (typecode 'B').
- ❷ Build memoryview from that array, then export it as a list.
- ❸ Build new memoryview from that previous one, but with 2 rows and 3 columns.
- ❹ Yet another memoryview, now with 3 rows and 2 columns.
- ❺ Overwrite byte in m2 at row 1, column 1 with 22.
- ❻ Overwrite byte in m3 at row 1, column 1 with 33.
- ❼ Display original array, proving that the memory was shared among octets, m1, m2, and m3.

The awesome power of memoryview can also be used to corrupt. [Example 2-21](#) shows how to change a single byte of an item in an array of 16-bit integers.

Example 2-21. Changing the value of a 16-bit integer array item by poking one of its bytes

```

>>> numbers = array.array('h', [-2, -1, 0, 1, 2])
>>> memv = memoryview(numbers) ❶
>>> len(memv)
5
>>> memv[0] ❷
-2
>>> memv_oct = memv.cast('B') ❸
>>> memv_oct.tolist() ❹
[254, 255, 255, 255, 0, 0, 1, 0, 2, 0]
>>> memv_oct[5] = 4 ❺
>>> numbers
array('h', [-2, -1, 1024, 1, 2]) ❻

```

- ❶ Build `memoryview` from array of 5 16-bit signed integers (typecode `'h'`).
- ❷ `memv` sees the same 5 items in the array.
- ❸ Create `memv_oct` by casting the elements of `memv` to bytes (typecode `'B'`).
- ❹ Export elements of `memv_oct` as a list of 10 bytes, for inspection.
- ❺ Assign value 4 to byte offset 5.
- ❻ Note the change to numbers: a 4 in the most significant byte of a 2-byte unsigned integer is 1024.



You'll find an example of inspecting `memoryview` with the `struct` package at fluentpython.com: “[Parsing binary records with struct](#)”.

Meanwhile, if you are doing advanced numeric processing in arrays, you should be using the NumPy libraries. We'll take a brief look at them right away.

NumPy

Throughout this book, I make a point of highlighting what is already in the Python standard library so you can make the most of it. But NumPy is so awesome that a detour is warranted.

For advanced array and matrix operations, NumPy is the reason why Python became mainstream in scientific computing applications. NumPy implements multi-dimensional, homogeneous arrays and matrix types that hold not only numbers but also user-defined records, and provides efficient element-wise operations.

SciPy is a library, written on top of NumPy, offering many scientific computing algorithms from linear algebra, numerical calculus, and statistics. SciPy is fast and reliable because it leverages the widely used C and Fortran codebase from the [Netlib Repository](#). In other words, SciPy gives scientists the best of both worlds: an interactive prompt and high-level Python APIs, together with industrial-strength number-crunching functions optimized in C and Fortran.

As a very brief NumPy demo, [Example 2-22](#) shows some basic operations with two-dimensional arrays.

Example 2-22. Basic operations with rows and columns in a `numpy.ndarray`

```
>>> import numpy as np ❶
>>> a = np.arange(12) ❷
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> type(a)
<class 'numpy.ndarray'>
>>> a.shape ❸
(12,)
>>> a.shape = 3, 4 ❹
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a[2] ❺
array([ 8,  9, 10, 11])
>>> a[2, 1] ❻
9
>>> a[:, 1] ❼
array([1, 5, 9])
>>> a.transpose() ❽
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

- ❶ Import NumPy, after installing (it's not in the Python standard library). Conventionally, `numpy` is imported as `np`.
- ❷ Build and inspect a `numpy.ndarray` with integers 0 to 11.
- ❸ Inspect the dimensions of the array: this is a one-dimensional, 12-element array.
- ❹ Change the shape of the array, adding one dimension, then inspecting the result.
- ❺ Get row at index 2.
- ❻ Get element at index 2, 1.
- ❼ Get column at index 1.
- ❽ Create a new array by transposing (swapping columns with rows).

NumPy also supports high-level operations for loading, saving, and operating on all elements of a `numpy.ndarray`:

```
>>> import numpy
>>> floats = numpy.loadtxt('floats-10M-lines.txt') ❶
```

```

>>> floats[-3:] ❷
array([ 3016362.69195522,  535281.10514262,  4566560.44373946])
>>> floats *= .5 ❸
>>> floats[-3:]
array([ 1508181.34597761,  267640.55257131,  2283280.22186973])
>>> from time import perf_counter as pc ❹
>>> t0 = pc(); floats /= 3; pc() - t0 ❺
0.03690556302899495
>>> numpy.save('floats-10M', floats) ❻
>>> floats2 = numpy.load('floats-10M.npy', 'r+') ❼
>>> floats2 *= 6
>>> floats2[-3:] ❽
memmap([ 3016362.69195522,  535281.10514262,  4566560.44373946])

```

- ❶ Load 10 million floating-point numbers from a text file.
- ❷ Use sequence slicing notation to inspect the last three numbers.
- ❸ Multiply every element in the `floats` array by `.5` and inspect the last three elements again.
- ❹ Import the high-resolution performance measurement timer (available since Python 3.3).
- ❺ Divide every element by 3; the elapsed time for 10 million floats is less than 40 milliseconds.
- ❻ Save the array in a `.npy` binary file.
- ❼ Load the data as a memory-mapped file into another array; this allows efficient processing of slices of the array even if it does not fit entirely in memory.
- ❽ Inspect the last three elements after multiplying every element by 6.

This was just an appetizer.

NumPy and SciPy are formidable libraries, and are the foundation of other awesome tools such as the **Pandas**—which implements efficient array types that can hold non-numeric data and provides import/export functions for many different formats, like `.csv`, `.xls`, SQL dumps, HDF5, etc.—and **scikit-learn**, currently the most widely used Machine Learning toolset. Most NumPy and SciPy functions are implemented in C or C++, and can leverage all CPU cores because they release Python’s GIL (Global Interpreter Lock). The **Dask** project supports parallelizing NumPy, Pandas, and scikit-learn processing across clusters of machines. These packages deserve entire books about them. This is not one of those books. But no overview of Python sequences would be complete without at least a quick look at NumPy arrays.

Having looked at flat sequences—standard arrays and NumPy arrays—we now turn to a completely different set of replacements for the plain old `list`: queues.

Dequeues and Other Queues

The `.append` and `.pop` methods make a `list` usable as a stack or a queue (if you use `.append` and `.pop(0)`, you get FIFO behavior). But inserting and removing from the head of a list (the 0-index end) is costly because the entire list must be shifted in memory.

The class `collections.deque` is a thread-safe double-ended queue designed for fast inserting and removing from both ends. It is also the way to go if you need to keep a list of “last seen items” or something of that nature, because a deque can be bounded—i.e., created with a fixed maximum length. If a bounded deque is full, when you add a new item, it discards an item from the opposite end. [Example 2-23](#) shows some typical operations performed on a deque.

Example 2-23. Working with a deque

```
>>> from collections import deque
>>> dq = deque(range(10), maxlen=10) ❶
>>> dq
deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.rotate(3) ❷
>>> dq
deque([7, 8, 9, 0, 1, 2, 3, 4, 5, 6], maxlen=10)
>>> dq.rotate(-4)
>>> dq
deque([1, 2, 3, 4, 5, 6, 7, 8, 9, 0], maxlen=10)
>>> dq.appendleft(-1) ❸
>>> dq
deque([-1, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.extend([11, 22, 33]) ❹
>>> dq
deque([3, 4, 5, 6, 7, 8, 9, 11, 22, 33], maxlen=10)
>>> dq.extendleft([10, 20, 30, 40]) ❺
>>> dq
deque([40, 30, 20, 10, 3, 4, 5, 6, 7, 8], maxlen=10)
```

- ❶ The optional `maxlen` argument sets the maximum number of items allowed in this instance of deque; this sets a read-only `maxlen` instance attribute.
- ❷ Rotating with `n > 0` takes items from the right end and prepends them to the left; when `n < 0` items are taken from left and appended to the right.

- ③ Appending to a deque that is full (`len(d) == d.maxlen`) discards items from the other end; note in the next line that the 0 is dropped.
- ④ Adding three items to the right pushes out the leftmost -1, 1, and 2.
- ⑤ Note that `extendleft(iter)` works by appending each successive item of the `iter` argument to the left of the deque, therefore the final position of the items is reversed.

Table 2-4 compares the methods that are specific to `list` and `deque` (removing those that also appear in `object`).

Note that `deque` implements most of the `list` methods, and adds a few that are specific to its design, like `popleft` and `rotate`. But there is a hidden cost: removing items from the middle of a deque is not as fast. It is really optimized for appending and popping from the ends.

The `append` and `popleft` operations are atomic, so `deque` is safe to use as a FIFO queue in multithreaded applications without the need for locks.

Table 2-4. Methods implemented in `list` or `deque` (those that are also implemented by `object` are omitted for brevity)

	list	deque	
<code>s.__add__(s2)</code>	•		<code>s + s2</code> —concatenation
<code>s.__iadd__(s2)</code>	•	•	<code>s += s2</code> —in-place concatenation
<code>s.append(e)</code>	•	•	Append one element to the right (after last)
<code>s.appendleft(e)</code>		•	Append one element to the left (before first)
<code>s.clear()</code>	•	•	Delete all items
<code>s.__contains__(e)</code>	•		<code>e in s</code>
<code>s.copy()</code>	•		Shallow copy of the list
<code>s.__copy__()</code>		•	Support for <code>copy.copy</code> (shallow copy)
<code>s.count(e)</code>	•	•	Count occurrences of an element
<code>s.__delitem__(p)</code>	•	•	Remove item at position <code>p</code>
<code>s.extend(i)</code>	•	•	Append items from iterable <code>i</code> to the right
<code>s.extendleft(i)</code>		•	Append items from iterable <code>i</code> to the left
<code>s.__getitem__(p)</code>	•	•	<code>s[p]</code> —get item or slice at position
<code>s.index(e)</code>	•		Find position of first occurrence of <code>e</code>
<code>s.insert(p, e)</code>	•		Insert element <code>e</code> before the item at position <code>p</code>
<code>s.__iter__()</code>	•	•	Get iterator
<code>s.__len__()</code>	•	•	<code>len(s)</code> —number of items

	list	deque
<code>s.__mul__(n)</code>	•	<code>s * n</code> —repeated concatenation
<code>s.__imul__(n)</code>	•	<code>s *= n</code> —in-place repeated concatenation
<code>s.__rmul__(n)</code>	•	<code>n * s</code> —reversed repeated concatenation ^a
<code>s.pop()</code>	• •	Remove and return last item ^b
<code>s.popleft()</code>	•	Remove and return first item
<code>s.remove(e)</code>	• •	Remove first occurrence of element <code>e</code> by value
<code>s.reverse()</code>	• •	Reverse the order of the items in place
<code>s.__reversed__()</code>	• •	Get iterator to scan items from last to first
<code>s.rotate(n)</code>	•	Move <code>n</code> items from one end to the other
<code>s.__setitem__(p, e)</code>	• •	<code>s[p] = e</code> —put <code>e</code> in position <code>p</code> , overwriting existing item or slice
<code>s.sort([key], [reverse])</code>	•	Sort items in place with optional keyword arguments <code>key</code> and <code>reverse</code>

^a Reversed operators are explained in [Chapter 16](#).

^b `a_list.pop(p)` allows removing from position `p`, but `deque` does not support that option.

Besides `deque`, other Python standard library packages implement queues:

queue

This provides the synchronized (i.e., thread-safe) classes `SimpleQueue`, `Queue`, `LifoQueue`, and `PriorityQueue`. These can be used for safe communication between threads. All except `SimpleQueue` can be bounded by providing a `max` size argument greater than 0 to the constructor. However, they don't discard items to make room as `deque` does. Instead, when the queue is full, the insertion of a new item blocks—i.e., it waits until some other thread makes room by taking an item from the queue, which is useful to throttle the number of live threads.

multiprocessing

Implements its own unbounded `SimpleQueue` and bounded `Queue`, very similar to those in the `queue` package, but designed for interprocess communication. A specialized `multiprocessing.JoinableQueue` is provided for task management.

asyncio

Provides `Queue`, `LifoQueue`, `PriorityQueue`, and `JoinableQueue` with APIs inspired by the classes in the `queue` and `multiprocessing` modules, but adapted for managing tasks in asynchronous programming.

heapq

In contrast to the previous three modules, `heapq` does not implement a queue class, but provides functions like `heappush` and `heappop` that let you use a mutable sequence as a heap queue or priority queue.

This ends our overview of alternatives to the `list` type, and also our exploration of sequence types in general—except for the particulars of `str` and binary sequences, which have their own chapter ([Chapter 4](#)).

Chapter Summary

Mastering the standard library sequence types is a prerequisite for writing concise, effective, and idiomatic Python code.

Python sequences are often categorized as mutable or immutable, but it is also useful to consider a different axis: flat sequences and container sequences. The former are more compact, faster, and easier to use, but are limited to storing atomic data such as numbers, characters, and bytes. Container sequences are more flexible, but may surprise you when they hold mutable objects, so you need to be careful to use them correctly with nested data structures.

Unfortunately, Python has no foolproof immutable container sequence type: even “immutable” tuples can have their values changed when they contain mutable items like lists or user-defined objects.

List comprehensions and generator expressions are powerful notations to build and initialize sequences. If you are not yet comfortable with them, take the time to master their basic usage. It is not hard, and soon you will be hooked.

Tuples in Python play two roles: as records with unnamed fields and as immutable lists. When using a tuple as an immutable list, remember that a tuple value is only guaranteed to be fixed if all the items in it are also immutable. Calling `hash(t)` on a tuple is a quick way to assert that its value is fixed. A `TypeError` will be raised if `t` contains mutable items.

When a tuple is used as a record, tuple unpacking is the safest, most readable way of extracting the fields of the tuple. Beyond tuples, `*` works with lists and iterables in many contexts, and some of its use cases appeared in Python 3.5 with [PEP 448—Additional Unpacking Generalizations](#). Python 3.10 introduced pattern matching with `match/case`, supporting more powerful unpacking, known as destructuring.

Sequence slicing is a favorite Python syntax feature, and it is even more powerful than many realize. Multidimensional slicing and ellipsis (`...`) notation, as used in NumPy, may also be supported by user-defined sequences. Assigning to slices is a very expressive way of editing mutable sequences.

Repeated concatenation as in `seq * n` is convenient and, with care, can be used to initialize lists of lists containing immutable items. Augmented assignment with `+=` and `*=` behaves differently for mutable and immutable sequences. In the latter case, these operators necessarily build new sequences. But if the target sequence is

mutable, it is usually changed in place—but not always, depending on how the sequence is implemented.

The `sort` method and the `sorted` built-in function are easy to use and flexible, thanks to the optional `key` argument: a function to calculate the ordering criterion. By the way, `key` can also be used with the `min` and `max` built-in functions.

Beyond lists and tuples, the Python standard library provides `array.array`. Although NumPy and SciPy are not part of the standard library, if you do any kind of numerical processing on large sets of data, studying even a small part of these libraries can take you a long way.

We closed by visiting the versatile and thread-safe `collections.deque`, comparing its API with that of `list` in [Table 2-4](#) and mentioning other queue implementations in the standard library.

Further Reading

Chapter 1, “Data Structures,” of the *Python Cookbook, 3rd ed.* (O’Reilly) by David Beazley and Brian K. Jones, has many recipes focusing on sequences, including “Recipe 1.11. Naming a Slice,” from which I learned the trick of assigning slices to variables to improve readability, illustrated in our [Example 2-13](#).

The second edition of the *Python Cookbook* was written for Python 2.4, but much of its code works with Python 3, and a lot of the recipes in Chapters 5 and 6 deal with sequences. The book was edited by Alex Martelli, Anna Ravenscroft, and David Ascher, and it includes contributions by dozens of Pythonistas. The third edition was rewritten from scratch, and focuses more on the semantics of the language—particularly what has changed in Python 3—while the older volume emphasizes pragmatics (i.e., how to apply the language to real-world problems). Even though some of the second edition solutions are no longer the best approach, I honestly think it is worthwhile to have both editions of the *Python Cookbook* on hand.

The official Python “[Sorting HOW TO](#)” has several examples of advanced tricks for using `sorted` and `list.sort`.

[PEP 3132—Extended Iterable Unpacking](#) is the canonical source to read about the new use of `*extra` syntax on the lefthand side of parallel assignments. If you’d like a glimpse of Python evolving, “[Missing *-unpacking generalizations](#)” is a bug tracker issue proposing enhancements to the iterable unpacking notation. [PEP 448—Additional Unpacking Generalizations](#) resulted from the discussions in that issue.

As I mentioned in “Pattern Matching with Sequences” on page 38, Carol Willing’s “Structural Pattern Matching” section of “What’s New In Python 3.10” is a great introduction to this major new feature in about 1,400 words (that’s less than 5 pages when Firefox makes a PDF from the HTML). PEP 636—Structural Pattern Matching: Tutorial is also good, but longer. The same PEP 636 includes “Appendix A—Quick Intro”. It is shorter than Willing’s intro because it omits high-level considerations about why pattern matching is good for you. If you need more arguments to convince yourself or others that pattern matching is good for Python, read the 22-page PEP 635—Structural Pattern Matching: Motivation and Rationale.

Eli Bendersky’s blog post “Less copies in Python with the buffer protocol and memoryviews” includes a short tutorial on memoryview.

There are numerous books covering NumPy in the market, and many don’t mention “NumPy” in the title. Two examples are the open access *Python Data Science Handbook* by Jake VanderPlas, and the second edition of Wes McKinney’s *Python for Data Analysis*.

“NumPy is all about vectorization.” That is the opening sentence of Nicolas P. Rougier’s open access book *From Python to NumPy*. Vectorized operations apply mathematical functions to all elements of an array without an explicit loop written in Python. They can operate in parallel, using special vector instructions in modern CPUs, leveraging multiple cores or delegating to the GPU, depending on the library. The first example in Rougier’s book shows a speedup of 500 times after refactoring a nice Pythonic class using a generator method, into a lean and mean function calling a couple of NumPy vector functions.

To learn how to use deque (and other collections), see the examples and practical recipes in “Container datatypes” in the Python documentation.

The best defense of the Python convention of excluding the last item in ranges and slices was written by Edsger W. Dijkstra himself, in a short memo titled “Why Numbering Should Start at Zero”. The subject of the memo is mathematical notation, but it’s relevant to Python because Dijkstra explains with rigor and humor why a sequence like 2, 3, ..., 12 should always be expressed as $2 \leq i < 13$. All other reasonable conventions are refuted, as is the idea of letting each user choose a convention. The title refers to zero-based indexing, but the memo is really about why it is desirable that 'ABCDE'[1:3] means 'BC' and not 'BCD' and why it makes perfect sense to write range(2, 13) to produce 2, 3, 4, ..., 12. By the way, the memo is a handwritten note, but it’s beautiful and totally readable. Dijkstra’s handwriting is so clear that someone created a font out of his notes.

Soapbox

The Nature of Tuples

In 2012, I presented a poster about the ABC language at PyCon US. Before creating Python, Guido van Rossum had worked on the ABC interpreter, so he came to see my poster. Among other things, we talked about the ABC *compounds*, which are clearly the predecessors of Python tuples. Compounds also support parallel assignment and are used as composite keys in dictionaries (or *tables*, in ABC parlance). However, compounds are not sequences. They are not iterable and you cannot retrieve a field by index, much less slice them. You either handle the compound as whole or extract the individual fields using parallel assignment, that's all.

I told Guido that these limitations make the main purpose of compounds very clear: they are just records without field names. His response: “Making tuples behave as sequences was a hack.”

This illustrates the pragmatic approach that made Python more practical and more successful than ABC. From a language implementer perspective, making tuples behave as sequences costs little. As a result, the main use case for tuples as records is not so obvious, but we gained immutable lists—even if their type is not as clearly named as `frozenset`.

Flat Versus Container Sequences

To highlight the different memory models of the sequence types, I used the terms *container sequence* and *flat sequence*. The “container” word is from [the “Data Model” documentation](#):

Some objects contain references to other objects; these are called containers.

I used the term “container sequence” to be specific, because there are containers in Python that are not sequences, like `dict` and `set`. Container sequences can be nested because they may contain objects of any type, including their own type.

On the other hand, *flat sequences* are sequence types that cannot be nested because they only hold simple atomic types like integers, floats, or characters.

I adopted the term *flat sequence* because I needed something to contrast with “container sequence.”

Despite the previous use of the word “containers” in the official documentation, there is an abstract class in `collections.abc` called `Container`. That ABC has just one method, `__contains__`—the special method behind the `in` operator. This means that strings and arrays, which are not containers in the traditional sense, are virtual subclasses of `Container` because they implement `__contains__`. This is just one more example of humans using a word to mean different things. In this book I'll write “container” with lowercase letters to mean “an object that contains references to

other objects,” and `Container` with a capitalized initial in a single-spaced font to refer to collections.`abc.Container`.

Mixed-Bag Lists

Introductory Python texts emphasize that lists can contain objects of mixed types, but in practice that feature is not very useful: we put items in a list to process them later, which implies that all items should support at least some operation in common (i.e., they should all “quack” whether or not they are genetically 100% ducks). For example, you can’t sort a list in Python 3 unless the items in it are comparable:

```
>>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
>>> sorted(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
```

Unlike lists, tuples often hold items of different types. That’s natural: if each item in a tuple is a field, then each field may have a different type.

key Is Brilliant

The optional `key` argument of `list.sort`, `sorted`, `max`, and `min` is a great idea. Other languages force you to provide a two-argument comparison function like the deprecated `cmp(a, b)` function in Python 2. Using `key` is both simpler and more efficient. It’s simpler because you just define a one-argument function that retrieves or calculates whatever criterion you want to use to sort your objects; this is easier than writing a two-argument function to return `-1`, `0`, `1`. It is also more efficient because the `key` function is invoked only once per item, while the two-argument comparison is called every time the sorting algorithm needs to compare two items. Of course, Python also has to compare the keys while sorting, but that comparison is done in optimized C code and not in a Python function that you wrote.

By the way, using `key` we can sort a mixed bag of numbers and number-like strings. We just need to decide whether we want to treat all items as integers or strings:

```
>>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
>>> sorted(l, key=int)
[0, '1', 5, 6, '9', 14, 19, '23', 28, '28']
>>> sorted(l, key=str)
[0, '1', 14, 19, '23', 28, '28', 5, 6, '9']
```

Oracle, Google, and the Timbot Conspiracy

The sorting algorithm used in `sorted` and `list.sort` is Timsort, an adaptive algorithm that switches from insertion sort to merge sort strategies, depending on how ordered the data is. This is efficient because real-world data tends to have runs of sorted items. There is a [Wikipedia article](#) about it.

Timsort was first used in CPython in 2002. Since 2009, Timsort is also used to sort arrays in both standard Java and Android, a fact that became widely known when Oracle used some of the code related to Timsort as evidence of Google infringement of Sun's intellectual property. For example, see this [order by Judge William Alsup](#) from 2012. In 2021, the US Supreme Court ruled Google's use of Java code as "fair use."

Timsort was invented by Tim Peters, a Python core developer so prolific that he is believed to be an AI, the Timbot. You can read about that conspiracy theory in ["Python Humor"](#). Tim also wrote "The Zen of Python": `import this`.

Dictionaries and Sets

Python is basically dicts wrapped in loads of syntactic sugar.

—Lalo Martins, early digital nomad and Pythonista

We use dictionaries in all our Python programs. If not directly in our code, then indirectly because the `dict` type is a fundamental part of Python’s implementation. Class and instance attributes, module namespaces, and function keyword arguments are some of the core Python constructs represented by dictionaries in memory. The `__builtins__.__dict__` stores all built-in types, objects, and functions.

Because of their crucial role, Python dicts are highly optimized—and continue to get improvements. *Hash tables* are the engines behind Python’s high-performance dicts.

Other built-in types based on hash tables are `set` and `frozenset`. These offer richer APIs and operators than the sets you may have encountered in other popular languages. In particular, Python sets implement all the fundamental operations from set theory, like union, intersection, subset tests, etc. With them, we can express algorithms in a more declarative way, avoiding lots of nested loops and conditionals.

Here is a brief outline of this chapter:

- Modern syntax to build and handle dicts and mappings, including enhanced unpacking and pattern matching
- Common methods of mapping types
- Special handling for missing keys
- Variations of `dict` in the standard library
- The `set` and `frozenset` types
- Implications of hash tables in the behavior of sets and dictionaries

What's New in This Chapter

Most changes in this second edition cover new features related to mapping types:

- “Modern dict Syntax” on page 78 covers enhanced unpacking syntax and different ways of merging mappings—including the `|` and `|=` operators supported by dicts since Python 3.9.
- “Pattern Matching with Mappings” on page 81 illustrates handling mappings with `match/case`, since Python 3.10.
- “`collections.OrderedDict`” on page 95 now focuses on the small but still relevant differences between `dict` and `OrderedDict`—considering that `dict` keeps the key insertion order since Python 3.6.
- New sections on the view objects returned by `dict.keys`, `dict.items`, and `dict.values`: “Dictionary Views” on page 101 and “Set Operations on dict Views” on page 110.

The underlying implementation of `dict` and `set` still relies on hash tables, but the `dict` code has two important optimizations that save memory and preserve the insertion order of the keys in `dict`. “Practical Consequences of How dict Works” on page 102 and “Practical Consequences of How Sets Work” on page 107 summarize what you need to know to use them well.



After adding more than 200 pages in this second edition, I moved the optional section “Internals of sets and dicts” to the *fluentpython.com* companion website. The updated and expanded 18-page post includes explanations and diagrams about:

- The hash table algorithm and data structures, starting with its use in `set`, which is simpler to understand.
- The memory optimization that preserves key insertion order in `dict` instances (since Python 3.6).
- The key-sharing layout for dictionaries holding instance attributes—the `__dict__` of user-defined objects (optimization implemented in Python 3.3).

Modern dict Syntax

The next sections describe advanced syntax features to build, unpack, and process mappings. Some of these features are not new in the language, but may be new to you. Others require Python 3.9 (like the `|` operator) or Python 3.10 (like `match/case`). Let's start with one of the best and oldest of these features.

dict Comprehensions

Since Python 2.7, the syntax of listcomps and genexps was adapted to dict comprehensions (and set comprehensions as well, which we'll soon visit). A *dictcomp* (dict comprehension) builds a dict instance by taking key:value pairs from any iterable.

Example 3-1 shows the use of dict comprehensions to build two dictionaries from the same list of tuples.

Example 3-1. Examples of dict comprehensions

```
>>> dial_codes = [
...     (880, 'Bangladesh'),
...     (55, 'Brazil'),
...     (86, 'China'),
...     (91, 'India'),
...     (62, 'Indonesia'),
...     (81, 'Japan'),
...     (234, 'Nigeria'),
...     (92, 'Pakistan'),
...     (7, 'Russia'),
...     (1, 'United States'),
... ]
>>> country_dial = {country: code for code, country in dial_codes}
>>> country_dial
{'Bangladesh': 880, 'Brazil': 55, 'China': 86, 'India': 91, 'Indonesia': 62,
'Japan': 81, 'Nigeria': 234, 'Pakistan': 92, 'Russia': 7, 'United States': 1}
>>> {code: country.upper()
...     for country, code in sorted(country_dial.items())
...     if code < 70}
{55: 'BRAZIL', 62: 'INDONESIA', 7: 'RUSSIA', 1: 'UNITED STATES'}
```

- ❶ An iterable of key-value pairs like `dial_codes` can be passed directly to the dict constructor, but...
- ❷ ...here we swap the pairs: `country` is the key, and `code` is the value.
- ❸ Sorting `country_dial` by name, reversing the pairs again, uppercasing values, and filtering items with `code < 70`.

If you're used to listcomps, dictcomps are a natural next step. If you aren't, the spread of the comprehension syntax means it's now more profitable than ever to become fluent in it.

Unpacking Mappings

PEP 448—[Additional Unpacking Generalizations](#) enhanced the support of mapping unpackings in two ways, since Python 3.5.

First, we can apply `**` to more than one argument in a function call. This works when keys are all strings and unique across all arguments (because duplicate keyword arguments are forbidden):

```
>>> def dump(**kwargs):
...     return kwargs
...
>>> dump(**{'x': 1}, y=2, **{'z': 3})
{'x': 1, 'y': 2, 'z': 3}
```

Second, `**` can be used inside a dict literal—also multiple times:

```
>>> {'a': 0, **{'x': 1}, 'y': 2, **{'z': 3, 'x': 4}}
{'a': 0, 'x': 4, 'y': 2, 'z': 3}
```

In this case, duplicate keys are allowed. Later occurrences overwrite previous ones—see the value mapped to `x` in the example.

This syntax can also be used to merge mappings, but there are other ways. Please read on.

Merging Mappings with |

Python 3.9 supports using `|` and `|=` to merge mappings. This makes sense, since these are also the set union operators.

The `|` operator creates a new mapping:

```
>>> d1 = {'a': 1, 'b': 3}
>>> d2 = {'a': 2, 'b': 4, 'c': 6}
>>> d1 | d2
{'a': 2, 'b': 4, 'c': 6}
```

Usually the type of the new mapping will be the same as the type of the left operand—`d1` in the example—but it can be the type of the second operand if user-defined types are involved, according to the operator overloading rules we explore in [Chapter 16](#).

To update an existing mapping in place, use `|=`. Continuing from the previous example, `d1` was not changed, but now it is:

```
>>> d1
{'a': 1, 'b': 3}
>>> d1 |= d2
>>> d1
{'a': 2, 'b': 4, 'c': 6}
```



If you need to maintain code to run on Python 3.8 or earlier, the “Motivation” section of [PEP 584—Add Union Operators To dict](#) provides a good summary of other ways to merge mappings.

Now let’s see how pattern matching applies to mappings.

Pattern Matching with Mappings

The `match/case` statement supports subjects that are mapping objects. Patterns for mappings look like `dict` literals, but they can match instances of any actual or virtual subclass of `collections.abc.Mapping`.¹

In [Chapter 2](#) we focused on sequence patterns only, but different types of patterns can be combined and nested. Thanks to destructuring, pattern matching is a powerful tool to process records structured like nested mappings and sequences, which we often need to read from JSON APIs and databases with semi-structured schemas, like MongoDB, EdgeDB, or PostgreSQL. [Example 3-2](#) demonstrates that. The simple type hints in `get_creators` make it clear that it takes a `dict` and returns a `list`.

Example 3-2. `creator.py`: `get_creators()` extracts names of creators from media records

```
def get_creators(record: dict) -> list:
    match record:
        case {'type': 'book', 'api': 2, 'authors': [*names]}: ❶
            return names
        case {'type': 'book', 'api': 1, 'author': name}: ❷
            return [name]
        case {'type': 'book'}: ❸
            raise ValueError(f"Invalid 'book' record: {record!r}")
        case {'type': 'movie', 'director': name}: ❹
            return [name]
        case _: ❺
            raise ValueError(f"Invalid record: {record!r}")
```

¹ A virtual subclass is any class registered by calling the `.register()` method of an ABC, as explained in “[A Virtual Subclass of an ABC](#)” on page 460. A type implemented via Python/C API is also eligible if a specific marker bit is set. See `Py_TPFLAGS_MAPPING`.

- ❶ Match any mapping with 'type': 'book', 'api': 2, and an 'authors' key mapped to a sequence. Return the items in the sequence, as a new list.
- ❷ Match any mapping with 'type': 'book', 'api': 1, and an 'author' key mapped to any object. Return the object inside a list.
- ❸ Any other mapping with 'type': 'book' is invalid, raise ValueError.
- ❹ Match any mapping with 'type': 'movie' and a 'director' key mapped to a single object. Return the object inside a list.
- ❺ Any other subject is invalid, raise ValueError.

Example 3-2 shows some useful practices for handling semi-structured data such as JSON records:

- Include a field describing the kind of record (e.g., 'type': 'movie')
- Include a field identifying the schema version (e.g., 'api': 2) to allow for future evolution of public APIs
- Have case clauses to handle invalid records of a specific type (e.g., 'book'), as well as a catch-all

Now let's see how `get_creators` handles some concrete doctests:

```
>>> b1 = dict(api=1, author='Douglas Hofstadter',
...           type='book', title='Gödel, Escher, Bach')
>>> get_creators(b1)
['Douglas Hofstadter']
>>> from collections import OrderedDict
>>> b2 = OrderedDict(api=2, type='book',
...                  title='Python in a Nutshell',
...                  authors='Martelli Ravenscroft Holden'.split())
>>> get_creators(b2)
['Martelli', 'Ravenscroft', 'Holden']
>>> get_creators({'type': 'book', 'pages': 770})
Traceback (most recent call last):
...
ValueError: Invalid 'book' record: {'type': 'book', 'pages': 770}
>>> get_creators('Spam, spam, spam')
Traceback (most recent call last):
...
ValueError: Invalid record: 'Spam, spam, spam'
```

Note that the order of the keys in the patterns is irrelevant, even if the subject is an `OrderedDict` as `b2`.

In contrast with sequence patterns, mapping patterns succeed on partial matches. In the doctests, the `b1` and `b2` subjects include a `'title'` key that does not appear in any `'book'` pattern, yet they match.

There is no need to use `**extra` to match extra key-value pairs, but if you want to capture them as a dict, you can prefix one variable with `**`. It must be the last in the pattern, and `**_` is forbidden because it would be redundant. A simple example:

```
>>> food = dict(category='ice cream', flavor='vanilla', cost=199)
>>> match food:
...     case {'category': 'ice cream', **details}:
...         print(f'Ice cream details: {details}')
...
Ice cream details: {'flavor': 'vanilla', 'cost': 199}
```

In “Automatic Handling of Missing Keys” on page 90 we’ll study `defaultdict` and other mappings where key lookups via `__getitem__` (i.e., `d[key]`) succeed because missing items are created on the fly. In the context of pattern matching, a match succeeds only if the subject already has the required keys at the top of the match statement.



The automatic handling of missing keys is not triggered because pattern matching always uses the `d.get(key, sentinel)` method —where the default `sentinel` is a special marker value that cannot occur in user data.

Moving on from syntax and structure, let’s study the API of mappings.

Standard API of Mapping Types

The `collections.abc` module provides the `Mapping` and `MutableMapping` ABCs describing the interfaces of `dict` and similar types. See [Figure 3-1](#).

The main value of the ABCs is documenting and formalizing the standard interfaces for mappings, and serving as criteria for `isinstance` tests in code that needs to support mappings in a broad sense:

```
>>> my_dict = {}
>>> isinstance(my_dict, abc.Mapping)
True
>>> isinstance(my_dict, abc.MutableMapping)
True
```



Using `isinstance` with an ABC is often better than checking whether a function argument is of the concrete dict type, because then alternative mapping types can be used. We'll discuss this in detail in [Chapter 13](#).

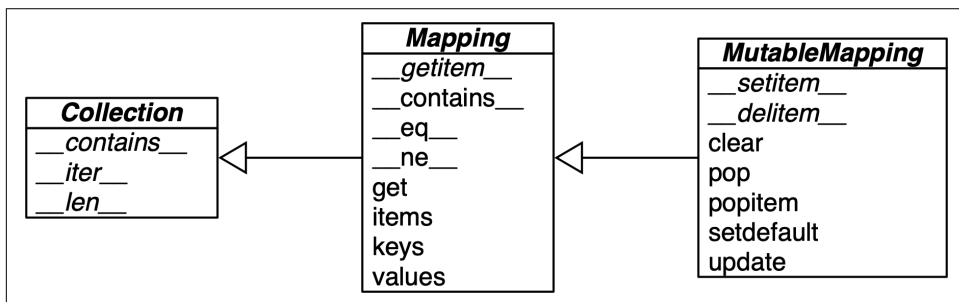


Figure 3-1. Simplified UML class diagram for the `MutableMapping` and its superclasses from `collections.abc` (inheritance arrows point from subclasses to superclasses; names in *italic* are abstract classes and abstract methods).

To implement a custom mapping, it's easier to extend `collections.UserDict`, or to wrap a `dict` by composition, instead of subclassing these ABCs. The `collections.UserDict` class and all concrete mapping classes in the standard library encapsulate the basic `dict` in their implementation, which in turn is built on a hash table. Therefore, they all share the limitation that the keys must be *hashable* (the values need not be hashable, only the keys). If you need a refresher, the next section explains.

What Is Hashable

Here is part of the definition of hashable adapted from the [Python Glossary](#):

An object is hashable if it has a hash code which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash code.²

Numeric types and flat immutable types `str` and `bytes` are all hashable. Container types are hashable if they are immutable and all contained objects are also hashable. A `frozenset` is always hashable, because every element it contains must be hashable

² The [Python Glossary](#) entry for “hashable” uses the term “hash value” instead of *hash code*. I prefer *hash code* because that is a concept often discussed in the context of mappings, where items are made of keys and values, so it may be confusing to mention the hash code as a value. In this book, I only use *hash code*.

by definition. A tuple is hashable only if all its items are hashable. See tuples `tt`, `tl`, and `tf`:

```
>>> tt = (1, 2, (30, 40))
>>> hash(tt)
8027212646858338501
>>> tl = (1, 2, [30, 40])
>>> hash(tl)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> tf = (1, 2, frozenset([30, 40]))
>>> hash(tf)
-4118419923444501110
```

The hash code of an object may be different depending on the version of Python, the machine architecture, and because of a *salt* added to the hash computation for security reasons.³ The hash code of a correctly implemented object is guaranteed to be constant only within one Python process.

User-defined types are hashable by default because their hash code is their `id()`, and the `__eq__()` method inherited from the object class simply compares the object IDs. If an object implements a custom `__eq__()` that takes into account its internal state, it will be hashable only if its `__hash__()` always returns the same hash code. In practice, this requires that `__eq__()` and `__hash__()` only take into account instance attributes that never change during the life of the object.

Now let's review the API of the most commonly used mapping types in Python: `dict`, `defaultdict`, and `OrderedDict`.

Overview of Common Mapping Methods

The basic API for mappings is quite rich. [Table 3-1](#) shows the methods implemented by `dict` and two popular variations: `defaultdict` and `OrderedDict`, both defined in the `collections` module.

³ See [PEP 456—Secure and interchangeable hash algorithm](#) to learn about the security implications and solutions adopted.

Table 3-1. Methods of the mapping types `dict`, `collections.defaultdict`, and `collections.OrderedDict` (common object methods omitted for brevity); optional arguments are enclosed in [...]

	<code>dict</code>	<code>defaultdict</code>	<code>OrderedDict</code>	
<code>d.clear()</code>	•	•	•	Remove all items
<code>d.__contains__(k)</code>	•	•	•	<code>k in d</code>
<code>d.copy()</code>	•	•	•	Shallow copy
<code>d.__copy__()</code>		•		Support for <code>copy.copy(d)</code>
<code>d.default_factory</code>		•		Callable invoked by <code>__missing__</code> to set missing values ^a
<code>d.__delitem__(k)</code>	•	•	•	<code>del d[k]</code> —remove item with key <code>k</code>
<code>d.fromkeys(it, [initial])</code>	•	•	•	New mapping from keys in iterable, with optional initial value (defaults to <code>None</code>)
<code>d.get(k, [default])</code>	•	•	•	Get item with key <code>k</code> , return <code>default</code> or <code>None</code> if missing
<code>d.__getitem__(k)</code>	•	•	•	<code>d[k]</code> —get item with key <code>k</code>
<code>d.items()</code>	•	•	•	Get <i>view</i> over items—(key, value) pairs
<code>d.__iter__()</code>	•	•	•	Get iterator over keys
<code>d.keys()</code>	•	•	•	Get <i>view</i> over keys
<code>d.__len__()</code>	•	•	•	<code>len(d)</code> —number of items
<code>d.__missing__(k)</code>		•		Called when <code>__getitem__</code> cannot find the key
<code>d.move_to_end(k, [last])</code>			•	Move <code>k</code> first or last position (<code>last</code> is <code>True</code> by default)
<code>d.__or__(other)</code>	•	•	•	Support for <code>d1 d2</code> to create new <code>dict</code> merging <code>d1</code> and <code>d2</code> (Python ≥ 3.9)
<code>d.__ior__(other)</code>	•	•	•	Support for <code>d1 = d2</code> to update <code>d1</code> with <code>d2</code> (Python ≥ 3.9)
<code>d.pop(k, [default])</code>	•	•	•	Remove and return value at <code>k</code> , or <code>default</code> or <code>None</code> if missing
<code>d.popitem()</code>	•	•	•	Remove and return the last inserted item as (key, value) ^b
<code>d.__reversed__()</code>	•	•	•	Support for <code>reverse(d)</code> —returns iterator for keys from last to first inserted.
<code>d.__ror__(other)</code>	•	•	•	Support for <code>other dd</code> —reversed union operator (Python ≥ 3.9) ^c

	dict	defaultdict	OrderedDict	
d.setdefault(k, [default])	•	•	•	If k in d, return d[k]; else set d[k] = default and return it
d.__setitem__(k, v)	•	•	•	d[k] = v—put v at k
d.update(m, **kwargs)	•	•	•	Update d with items from mapping or iterable of (key, value) pairs
d.values()	•	•	•	Get view over values

^a default_factory is not a method, but a callable attribute set by the end user when a defaultdict is instantiated.

^b OrderedDict.popitem(last=False) removes the first item inserted (FIFO). The last keyword argument is not supported in dict or defaultdict as recently as Python 3.10b3.

^c Reversed operators are explained in [Chapter 16](#).

The way d.update(m) handles its first argument m is a prime example of *duck typing*: it first checks whether m has a keys method and, if it does, assumes it is a mapping. Otherwise, update() falls back to iterating over m, assuming its items are (key, value) pairs. The constructor for most Python mappings uses the logic of update() internally, which means they can be initialized from other mappings or from any iterable object producing (key, value) pairs.

A subtle mapping method is setdefault(). It avoids redundant key lookups when we need to update the value of an item in place. The next section shows how to use it.

Inserting or Updating Mutable Values

In line with Python’s *fail-fast* philosophy, dict access with d[k] raises an error when k is not an existing key. Pythonistas know that d.get(k, default) is an alternative to d[k] whenever a default value is more convenient than handling KeyError. However, when you retrieve a mutable value and want to update it, there is a better way.

Consider a script to index text, producing a mapping where each key is a word, and the value is a list of positions where that word occurs, as shown in [Example 3-3](#).

Example 3-3. Partial output from [Example 3-4](#) processing the “Zen of Python”; each line shows a word and a list of occurrences coded as pairs: (line_number, column_number)

```
$ python3 index0.py zen.txt
a [(19, 48), (20, 53)]
Although [(11, 1), (16, 1), (18, 1)]
ambiguity [(14, 16)]
and [(15, 23)]
are [(21, 12)]
aren [(10, 15)]
at [(16, 38)]
bad [(19, 50)]
```

```

be [(15, 14), (16, 27), (20, 50)]
beats [(11, 23)]
Beautiful [(3, 1)]
better [(3, 14), (4, 13), (5, 11), (6, 12), (7, 9), (8, 11), (17, 8), (18, 25)]
...

```

Example 3-4 is a suboptimal script written to show one case where `dict.get` is not the best way to handle a missing key. I adapted it from an example by Alex Martelli.⁴

Example 3-4. `index0.py` uses `dict.get` to fetch and update a list of word occurrences from the index (a better solution is in [Example 3-5](#))

```

"""Build an index mapping word -> list of occurrences"""

import re
import sys

WORD_RE = re.compile(r'\w+')

index = {}
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start() + 1
            location = (line_no, column_no)
            # this is ugly; coded like this to make a point
            occurrences = index.get(word, []) ❶
            occurrences.append(location)       ❷
            index[word] = occurrences        ❸

# display in alphabetical order
for word in sorted(index, key=str.upper): ❹
    print(word, index[word])

```

- ❶ Get the list of occurrences for word, or `[]` if not found.
- ❷ Append new location to occurrences.
- ❸ Put changed occurrences into index dict; this entails a second search through the index.

⁴ The original script appears in slide 41 of Martelli's “[Re-learning Python](#)” presentation. His script is actually a demonstration of `dict.setdefault`, as shown in our [Example 3-5](#).

- ④ In the `key=` argument of `sorted`, I am not calling `str.upper`, just passing a reference to that method so the `sorted` function can use it to normalize the words for sorting.⁵

The three lines dealing with occurrences in [Example 3-4](#) can be replaced by a single line using `dict.setdefault`. [Example 3-5](#) is closer to Alex Martelli's code.

Example 3-5. `index.py` uses `dict.setdefault` to fetch and update a list of word occurrences from the index in a single line; contrast with [Example 3-4](#)

```
"""Build an index mapping word -> list of occurrences"""
```

```
import re
import sys

WORD_RE = re.compile(r'\w+')

index = {}
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start() + 1
            location = (line_no, column_no)
            index.setdefault(word, []).append(location) ❶

# display in alphabetical order
for word in sorted(index, key=str.upper):
    print(word, index[word])
```

- ❶ Get the list of occurrences for `word`, or set it to `[]` if not found; `setdefault` returns the value, so it can be updated without requiring a second search.

In other words, the end result of this line...

```
my_dict.setdefault(key, []).append(new_value)
```

...is the same as running...

```
if key not in my_dict:
    my_dict[key] = []
my_dict[key].append(new_value)
```

...except that the latter code performs at least two searches for `key`—three if it's not found—while `setdefault` does it all with a single lookup.

⁵ This is an example of using a method as a first-class function, the subject of [Chapter 7](#).

A related issue, handling missing keys on any lookup (and not only when inserting), is the subject of the next section.

Automatic Handling of Missing Keys

Sometimes it is convenient to have mappings that return some made-up value when a missing key is searched. There are two main approaches to this: one is to use a `defaultdict` instead of a plain `dict`. The other is to subclass `dict` or any other mapping type and add a `__missing__` method. Both solutions are covered next.

`defaultdict`: Another Take on Missing Keys

A `collections.defaultdict` instance creates items with a default value on demand whenever a missing key is searched using `d[k]` syntax. [Example 3-6](#) uses `defaultdict` to provide another elegant solution to the word index task from [Example 3-5](#).

Here is how it works: when instantiating a `defaultdict`, you provide a callable to produce a default value whenever `__getitem__` is passed a nonexistent key argument.

For example, given a `defaultdict` created as `dd = defaultdict(list)`, if `'new-key'` is not in `dd`, the expression `dd['new-key']` does the following steps:

1. Calls `list()` to create a new list.
2. Inserts the list into `dd` using `'new-key'` as key.
3. Returns a reference to that list.

The callable that produces the default values is held in an instance attribute named `default_factory`.

Example 3-6. `index_default.py`: using `defaultdict` instead of the `setdefault` method

```
"""Build an index mapping word -> list of occurrences"""
```

```
import collections
import re
import sys

WORD_RE = re.compile(r'\w+')

index = collections.defaultdict(list)  ❶
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start() + 1
            location = (line_no, column_no)
```



```

        index[word].append(location) ❷

# display in alphabetical order
for word in sorted(index, key=str.upper):
    print(word, index[word])

```

- ❶ Create a defaultdict with the list constructor as default_factory.
- ❷ If word is not initially in the index, the default_factory is called to produce the missing value, which in this case is an empty list that is then assigned to index[word] and returned, so the .append(location) operation always succeeds.

If no default_factory is provided, the usual KeyError is raised for missing keys.



The default_factory of a defaultdict is only invoked to provide default values for `__getitem__` calls, and not for the other methods. For example, if `dd` is a defaultdict, and `k` is a missing key, `dd[k]` will call the default_factory to create a default value, but `dd.get(k)` still returns `None`, and `k in dd` is `False`.

The mechanism that makes defaultdict work by calling default_factory is the `__missing__` special method, a feature that we discuss next.

The `__missing__` Method

Underlying the way mappings deal with missing keys is the aptly named `__missing__` method. This method is not defined in the base dict class, but dict is aware of it: if you subclass dict and provide a `__missing__` method, the standard dict.`__getitem__` will call it whenever a key is not found, instead of raising `KeyError`.

Suppose you'd like a mapping where keys are converted to str when looked up. A concrete use case is a device library for IoT,⁶ where a programmable board with general-purpose I/O pins (e.g., a Raspberry Pi or an Arduino) is represented by a Board class with a `my_board.pins` attribute, which is a mapping of physical pin identifiers to pin software objects. The physical pin identifier may be just a number or a string like "A0" or "P9_12". For consistency, it is desirable that all keys in `board.pins` are strings, but it is also convenient looking up a pin by number, as in `my_arduino.pin[13]`, so that beginners are not tripped when they want to blink the LED on pin 13 of their Arduinos. [Example 3-7](#) shows how such a mapping would work.

⁶ One such library is [Pingo.io](#), no longer under active development.

Example 3-7. When searching for a nonstring key, StrKeyDict0 converts it to str when it is not found

Tests for item retrieval using `d[key]` notation::

```
>>> d = StrKeyDict0([('2', 'two'), ('4', 'four')])
>>> d['2']
'two'
>>> d[4]
'four'
>>> d[1]
Traceback (most recent call last):
...
KeyError: '1'
```

Tests for item retrieval using `d.get(key)` notation::

```
>>> d.get('2')
'two'
>>> d.get(4)
'four'
>>> d.get(1, 'N/A')
'N/A'
```

Tests for the `in` operator::

```
>>> 2 in d
True
>>> 1 in d
False
```

Example 3-8 implements a class `StrKeyDict0` that passes the preceding doctests.



A better way to create a user-defined mapping type is to subclass `collections.UserDict` instead of `dict` (as we will do in [Example 3-9](#)). Here we subclass `dict` just to show that `__missing__` is supported by the built-in `dict.__getitem__` method.

Example 3-8. StrKeyDict0 converts nonstring keys to str on lookup (see tests in [Example 3-7](#))

```
class StrKeyDict0(dict): ❶

    def __missing__(self, key):
        if isinstance(key, str): ❷
            raise KeyError(key)
        return self[str(key)] ❸
```

```

def get(self, key, default=None):
    try:
        return self[key] ❹
    except KeyError:
        return default ❺

def __contains__(self, key):
    return key in self.keys() or str(key) in self.keys() ❻

```

- ❶ StrKeyDict0 inherits from dict.
- ❷ Check whether key is already a str. If it is, and it's missing, raise KeyError.
- ❸ Build str from key and look it up.
- ❹ The get method delegates to `__getitem__` by using the `self[key]` notation; that gives the opportunity for our `__missing__` to act.
- ❺ If a `KeyError` was raised, `__missing__` already failed, so we return the default.
- ❻ Search for unmodified key (the instance may contain non-str keys), then for a str built from the key.

Take a moment to consider why the test `isinstance(key, str)` is necessary in the `__missing__` implementation.

Without that test, our `__missing__` method would work OK for any key `k`—str or not str—whenever `str(k)` produced an existing key. But if `str(k)` is not an existing key, we'd have an infinite recursion. In the last line of `__missing__`, `self[str(key)]` would call `__getitem__`, passing that str key, which in turn would call `__missing__` again.

The `__contains__` method is also needed for consistent behavior in this example, because the operation `k in d` calls it, but the method inherited from `dict` does not fall back to invoking `__missing__`. There is a subtle detail in our implementation of `__contains__`: we do not check for the key in the usual Pythonic way—`k in my_dict`—because `str(key) in self` would recursively call `__contains__`. We avoid this by explicitly looking up the key in `self.keys()`.

A search like `k in my_dict.keys()` is efficient in Python 3 even for very large mappings because `dict.keys()` returns a view, which is similar to a set, as we'll see in “[Set Operations on dict Views](#)” on page 110. However, remember that `k in my_dict` does the same job, and is faster because it avoids the attribute lookup to find the `.keys` method.

I had a specific reason to use `self.keys()` in the `__contains__` method in **Example 3-8**. The check for the unmodified key—`key in self.keys()`—is necessary for correctness because `StrKeyDict0` does not enforce that all keys in the dictionary must be of type `str`. Our only goal with this simple example is to make searching “friendlier” and not enforce types.



User-defined classes derived from standard library mappings may or may not use `__missing__` as a fallback in their implementations of `__getitem__`, `get`, or `__contains__`, as explained in the next section.

Inconsistent Usage of `__missing__` in the Standard Library

Consider the following scenarios, and how the missing key lookups are affected:

dict subclass

A subclass of `dict` implementing only `__missing__` and no other method. In this case, `__missing__` may be called only on `d[k]`, which will use the `__getitem__` inherited from `dict`.

collections.UserDict subclass

Likewise, a subclass of `UserDict` implementing only `__missing__` and no other method. The `get` method inherited from `UserDict` calls `__getitem__`. This means `__missing__` may be called to handle lookups with `d[k]` and `d.get(k)`.

abc.Mapping subclass with the simplest possible __getitem__

A minimal subclass of `abc.Mapping` implementing `__missing__` and the required abstract methods, including an implementation of `__getitem__` that does not call `__missing__`. The `__missing__` method is never triggered in this class.

abc.Mapping subclass with __getitem__ calling __missing__

A minimal subclass of `abc.Mapping` implementing `__missing__` and the required abstract methods, including an implementation of `__getitem__` that calls `__missing__`. The `__missing__` method is triggered in this class for missing key lookups made with `d[k]`, `d.get(k)`, and `k in d`.

See *missing.py* in the example code repository for demonstrations of the scenarios described here.

The four scenarios just described assume minimal implementations. If your subclass implements `__getitem__`, `get`, and `__contains__`, then you can make those methods use `__missing__` or not, depending on your needs. The point of this section is to show that you must be careful when subclassing standard library mappings to use `__missing__`, because the base classes support different behaviors by default.

Don't forget that the behavior of `setdefault` and `update` is also affected by key lookup. And finally, depending on the logic of your `__missing__`, you may need to implement special logic in `__setitem__`, to avoid inconsistent or surprising behavior. We'll see an example of this in [“Subclassing UserDict Instead of dict” on page 97](#).

So far we have covered the `dict` and `defaultdict` mapping types, but the standard library comes with other mapping implementations, which we discuss next.

Variations of dict

In this section is an overview of mapping types included in the standard library, besides `defaultdict`, already covered in [“defaultdict: Another Take on Missing Keys” on page 90](#).

`collections.OrderedDict`

Now that the built-in `dict` also keeps the keys ordered since Python 3.6, the most common reason to use `OrderedDict` is writing code that is backward compatible with earlier Python versions. Having said that, Python's documentation lists some remaining differences between `dict` and `OrderedDict`, which I quote here—only reordering the items for relevance in daily use:

- The equality operation for `OrderedDict` checks for matching order.
- The `popitem()` method of `OrderedDict` has a different signature. It accepts an optional argument to specify which item is popped.
- `OrderedDict` has a `move_to_end()` method to efficiently reposition an element to an endpoint.
- The regular `dict` was designed to be very good at mapping operations. Tracking insertion order was secondary.
- `OrderedDict` was designed to be good at reordering operations. Space efficiency, iteration speed, and the performance of update operations were secondary.
- Algorithmically, `OrderedDict` can handle frequent reordering operations better than `dict`. This makes it suitable for tracking recent accesses (for example, in an LRU cache).

`collections.ChainMap`

A `ChainMap` instance holds a list of mappings that can be searched as one. The lookup is performed on each input mapping in the order it appears in the constructor call, and succeeds as soon as the key is found in one of those mappings. For example:

```

>>> d1 = dict(a=1, b=3)
>>> d2 = dict(a=2, b=4, c=6)
>>> from collections import ChainMap
>>> chain = ChainMap(d1, d2)
>>> chain['a']
1
>>> chain['c']
6

```

The ChainMap instance does not copy the input mappings, but holds references to them. Updates or insertions to a ChainMap only affect the first input mapping. Continuing from the previous example:

```

>>> chain['c'] = -1
>>> d1
{'a': 1, 'b': 3, 'c': -1}
>>> d2
{'a': 2, 'b': 4, 'c': 6}

```

ChainMap is useful to implement interpreters for languages with nested scopes, where each mapping represents a scope context, from the innermost enclosing scope to the outermost scope. The “ChainMap objects” section of the [collections docs](#) has several examples of ChainMap usage, including this snippet inspired by the basic rules of variable lookup in Python:

```

import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))

```

[Example 18-14](#) shows a ChainMap subclass used to implement an interpreter for a subset of the Scheme programming language.

collections.Counter

A mapping that holds an integer count for each key. Updating an existing key adds to its count. This can be used to count instances of hashable objects or as a multiset (discussed later in this section). Counter implements the + and - operators to combine tallies, and other useful methods such as `most_common([n])`, which returns an ordered list of tuples with the *n* most common items and their counts; see the [documentation](#). Here is Counter used to count letters in words:

```

>>> ct = collections.Counter('abracadabra')
>>> ct
Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
>>> ct.update('aaaazzz')
>>> ct
Counter({'a': 10, 'z': 3, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
>>> ct.most_common(3)
[('a', 10), ('z', 3), ('b', 2)]

```

Note that the 'b' and 'r' keys are tied in third place, but `ct.most_common(3)` shows only three counts.

To use `collections.Counter` as a multiset, pretend each key is an element in the set, and the count is the number of occurrences of that element in the set.

shelve.Shelf

The `shelve` module in the standard library provides persistent storage for a mapping of string keys to Python objects serialized in the `pickle` binary format. The curious name of `shelve` makes sense when you realize that pickle jars are stored on shelves.

The `shelve.open` module-level function returns a `shelve.Shelf` instance—a simple key-value DBM database backed by the `dbm` module, with these characteristics:

- `shelve.Shelf` subclasses `abc.MutableMapping`, so it provides the essential methods we expect of a mapping type.
- In addition, `shelve.Shelf` provides a few other I/O management methods, like `sync` and `close`.
- A `Shelf` instance is a context manager, so you can use a `with` block to make sure it is closed after use.
- Keys and values are saved whenever a new value is assigned to a key.
- The keys must be strings.
- The values must be objects that the `pickle` module can serialize.

The documentation for the `shelve`, `dbm`, and `pickle` modules provides more details and some caveats.



Python's `pickle` is easy to use in the simplest cases, but has several drawbacks. Read Ned Batchelder's "[Pickle's nine flaws](#)" before adopting any solution involving `pickle`. In his post, Ned mentions other serialization formats to consider.

`OrderedDict`, `ChainMap`, `Counter`, and `Shelf` are ready to use but can also be customized by subclassing. In contrast, `UserDict` is intended only as a base class to be extended.

Subclassing UserDict Instead of dict

It's better to create a new mapping type by extending `collections.UserDict` rather than `dict`. We realize that when we try to extend our `StrKeyDict0` from [Example 3-8](#) to make sure that any keys added to the mapping are stored as `str`.

The main reason why it's better to subclass `UserDict` rather than `dict` is that the built-in has some implementation shortcuts that end up forcing us to override methods that we can just inherit from `UserDict` with no problems.⁷

Note that `UserDict` does not inherit from `dict`, but uses composition: it has an internal `dict` instance, called `data`, which holds the actual items. This avoids undesired recursion when coding special methods like `__setitem__`, and simplifies the coding of `__contains__`, compared to [Example 3-8](#).

Thanks to `UserDict`, `StrKeyDict` ([Example 3-9](#)) is more concise than `StrKeyDict0` ([Example 3-8](#)), but it does more: it stores all keys as `str`, avoiding unpleasant surprises if the instance is built or updated with data containing nonstring keys.

Example 3-9. `StrKeyDict` always converts nonstring keys to `str` on insertion, update, and lookup

```
import collections

class StrKeyDict(collections.UserDict): ❶

    def __missing__(self, key): ❷
        if isinstance(key, str):
            raise KeyError(key)
        return self[str(key)]

    def __contains__(self, key):
        return str(key) in self.data ❸

    def __setitem__(self, key, item):
        self.data[str(key)] = item ❹
```

- ❶ `StrKeyDict` extends `UserDict`.
- ❷ `__missing__` is exactly as in [Example 3-8](#).
- ❸ `__contains__` is simpler: we can assume all stored keys are `str`, and we can check on `self.data` instead of invoking `self.keys()` as we did in `StrKeyDict0`.
- ❹ `__setitem__` converts any key to a `str`. This method is easier to overwrite when we can delegate to the `self.data` attribute.

⁷ The exact problem with subclassing `dict` and other built-ins is covered in “Subclassing Built-In Types Is Tricky” on page 490.

Because `UserDict` extends `abc.MutableMapping`, the remaining methods that make `StrKeyDict` a full-fledged mapping are inherited from `UserDict`, `MutableMapping`, or `Mapping`. The latter have several useful concrete methods, in spite of being abstract base classes (ABCs). The following methods are worth noting:

`MutableMapping.update`

This powerful method can be called directly but is also used by `__init__` to load the instance from other mappings, from iterables of `(key, value)` pairs, and keyword arguments. Because it uses `self[key] = value` to add items, it ends up calling our implementation of `__setitem__`.

`Mapping.get`

In `StrKeyDict0` (Example 3-8), we had to code our own `get` to return the same results as `__getitem__`, but in Example 3-9 we inherited `Mapping.get`, which is implemented exactly like `StrKeyDict0.get` (see the [Python source code](#)).



Antoine Pitrou authored [PEP 455—Adding a key-transforming dictionary to collections](#) and a patch to enhance the `collections` module with a `TransformDict`, that is more general than `StrKeyDict` and preserves the keys as they are provided, before the transformation is applied. PEP 455 was rejected in May 2015—see Raymond Hettinger’s [rejection message](#). To experiment with `TransformDict`, I extracted Pitrou’s patch from [issue18986](#) into a stand-alone module ([03-dict-set/transformdict.py](#) in the [Fluent Python second edition code repository](#)).

We know there are immutable sequence types, but how about an immutable mapping? Well, there isn’t a real one in the standard library, but a stand-in is available. That’s next.

Immutable Mappings

The mapping types provided by the standard library are all mutable, but you may need to prevent users from changing a mapping by accident. A concrete use case can be found, again, in a hardware programming library like *Pingo*, mentioned in “[The __missing__ Method](#)” on page 91: the `board.pins` mapping represents the physical GPIO pins on the device. As such, it’s useful to prevent inadvertent updates to `board.pins` because the hardware can’t be changed via software, so any change in the mapping would make it inconsistent with the physical reality of the device.

The `types` module provides a wrapper class called `MappingProxyType`, which, given a mapping, returns a `mappingproxy` instance that is a read-only but dynamic proxy for the original mapping. This means that updates to the original mapping can be seen in

the `mappingproxy`, but changes cannot be made through it. See [Example 3-10](#) for a brief demonstration.

Example 3-10. `MappingProxyType` builds a read-only `mappingproxy` instance from a `dict`

```
>>> from types import MappingProxyType
>>> d = {1: 'A'}
>>> d_proxy = MappingProxyType(d)
>>> d_proxy
mappingproxy({1: 'A'})
>>> d_proxy[1] ❶
'A'
>>> d_proxy[2] = 'x' ❷
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'mappingproxy' object does not support item assignment
>>> d[2] = 'B'
>>> d_proxy ❸
mappingproxy({1: 'A', 2: 'B'})
>>> d_proxy[2]
'B'
>>>
```

- ❶ Items in `d` can be seen through `d_proxy`.
- ❷ Changes cannot be made through `d_proxy`.
- ❸ `d_proxy` is dynamic: any change in `d` is reflected.

Here is how this could be used in practice in the hardware programming scenario: the constructor in a concrete `Board` subclass would fill a private mapping with the pin objects, and expose it to clients of the API via a public `.pins` attribute implemented as a `mappingproxy`. That way the clients would not be able to add, remove, or change pins by accident.

Next, we'll cover views—which allow high-performance operations on a `dict`, without unnecessary copying of data.

Dictionary Views

The dict instance methods `.keys()`, `.values()`, and `.items()` return instances of classes called `dict_keys`, `dict_values`, and `dict_items`, respectively. These dictionary views are read-only projections of the internal data structures used in the dict implementation. They avoid the memory overhead of the equivalent Python 2 methods that returned lists duplicating data already in the target dict, and they also replace the old methods that returned iterators.

Example 3-11 shows some basic operations supported by all dictionary views.

Example 3-11. The `.values()` method returns a view of the values in a dict

```
>>> d = dict(a=10, b=20, c=30)
>>> values = d.values()
>>> values
dict_values([10, 20, 30]) ❶
>>> len(values) ❷
3
>>> list(values) ❸
[10, 20, 30]
>>> reversed(values) ❹
<dict_reversevalueiterator object at 0x10e9e7310>
>>> values[0] ❺
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_values' object is not subscriptable
```

- ❶ The repr of a view object shows its content.
- ❷ We can query the len of a view.
- ❸ Views are iterable, so it's easy to create lists from them.
- ❹ Views implement `__reversed__`, returning a custom iterator.
- ❺ We can't use `[]` to get individual items from a view.

A view object is a dynamic proxy. If the source dict is updated, you can immediately see the changes through an existing view. Continuing from **Example 3-11**:

```
>>> d['z'] = 99
>>> d
{'a': 10, 'b': 20, 'c': 30, 'z': 99}
>>> values
dict_values([10, 20, 30, 99])
```

The classes `dict_keys`, `dict_values`, and `dict_items` are internal: they are not available via `__builtins__` or any standard library module, and even if you get a reference to one of them, you can't use it to create a view from scratch in Python code:

```
>>> values_class = type({}.values())
>>> v = values_class()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot create 'dict_values' instances
```

The `dict_values` class is the simplest dictionary view—it implements only the `__len__`, `__iter__`, and `__reversed__` special methods. In addition to these methods, `dict_keys` and `dict_items` implement several set methods, almost as many as the `frozenset` class. After we cover sets, we'll have more to say about `dict_keys` and `dict_items` in [“Set Operations on dict Views” on page 110](#).

Now let's see some rules and tips informed by the way `dict` is implemented under the hood.

Practical Consequences of How dict Works

The hash table implementation of Python's `dict` is very efficient, but it's important to understand the practical effects of this design:

- Keys must be hashable objects. They must implement proper `__hash__` and `__eq__` methods as described in [“What Is Hashable” on page 84](#).
- Item access by key is very fast. A `dict` may have millions of keys, but Python can locate a key directly by computing the hash code of the key and deriving an index offset into the hash table, with the possible overhead of a small number of tries to find a matching entry.
- Key ordering is preserved as a side effect of a more compact memory layout for `dict` in CPython 3.6, which became an official language feature in 3.7.
- Despite its new compact layout, `dicts` inevitably have a significant memory overhead. The most compact internal data structure for a container would be an array of pointers to the items.⁸ Compared to that, a hash table needs to store more data per entry, and Python needs to keep at least one-third of the hash table rows empty to remain efficient.
- To save memory, avoid creating instance attributes outside of the `__init__` method.

⁸ That's how tuples are stored.

That last tip about instance attributes comes from the fact that Python’s default behavior is to store instance attributes in a special `__dict__` attribute, which is a dict attached to each instance.⁹ Since [PEP 412—Key-Sharing Dictionary](#) was implemented in Python 3.3, instances of a class can share a common hash table, stored with the class. That common hash table is shared by the `__dict__` of each new instance that has the same attributes names as the first instance of that class when `__init__` returns. Each instance `__dict__` can then hold only its own attribute values as a simple array of pointers. Adding an instance attribute after `__init__` forces Python to create a new hash table just for the `__dict__` of that one instance (which was the default behavior for all instances before Python 3.3). According to PEP 412, this optimization reduces memory use by 10% to 20% for object-oriented programs.

The details of the compact layout and key-sharing optimizations are rather complex. For more, please read “[Internals of sets and dicts](#)” at [fluentpython.com](#).

Now let’s dive into sets.

Set Theory

Sets are not new in Python, but are still somewhat underused. The `set` type and its immutable sibling `frozenset` first appeared as modules in the Python 2.3 standard library, and were promoted to built-ins in Python 2.6.



In this book, I use the word “set” to refer both to `set` and `frozenset`. When talking specifically about the `set` class, I use constant width font: `set`.

A set is a collection of unique objects. A basic use case is removing duplication:

```
>>> l = ['spam', 'spam', 'eggs', 'spam', 'bacon', 'eggs']
>>> set(l)
{'eggs', 'spam', 'bacon'}
>>> list(set(l))
['eggs', 'spam', 'bacon']
```

⁹ Unless the class has a `__slots__` attribute, as explained in “[Saving Memory with `__slots__`](#)” on page 384.



If you want to remove duplicates but also preserve the order of the first occurrence of each item, you can now use a plain dict to do it, like this:

```
>>> dict.fromkeys(l).keys()
dict_keys(['spam', 'eggs', 'bacon'])
>>> list(dict.fromkeys(l).keys())
['spam', 'eggs', 'bacon']
```

Set elements must be hashable. The set type is not hashable, so you can't build a set with nested set instances. But frozenset is hashable, so you can have frozenset elements inside a set.

In addition to enforcing uniqueness, the set types implement many set operations as infix operators, so, given two sets *a* and *b*, *a* | *b* returns their union, *a* & *b* computes the intersection, *a* - *b* the difference, and *a* ^ *b* the symmetric difference. Smart use of set operations can reduce both the line count and the execution time of Python programs, at the same time making code easier to read and reason about—by removing loops and conditional logic.

For example, imagine you have a large set of email addresses (the haystack) and a smaller set of addresses (the needles) and you need to count how many needles occur in the haystack. Thanks to set intersection (the & operator) you can code that in a simple line (see [Example 3-12](#)).

Example 3-12. Count occurrences of needles in a haystack, both of type set

```
found = len(needles & haystack)
```

Without the intersection operator, you'd have to write [Example 3-13](#) to accomplish the same task as [Example 3-12](#).

Example 3-13. Count occurrences of needles in a haystack (same end result as [Example 3-12](#))

```
found = 0
for n in needles:
    if n in haystack:
        found += 1
```

[Example 3-12](#) runs slightly faster than [Example 3-13](#). On the other hand, [Example 3-13](#) works for any iterable objects *needles* and *haystack*, while [Example 3-12](#) requires that both be sets. But, if you don't have sets on hand, you can always build them on the fly, as shown in [Example 3-14](#).

Example 3-14. Count occurrences of needles in a haystack; these lines work for any iterable types

```
found = len(set(needles) & set(haystack))

# another way:
found = len(set(needles).intersection(haystack))
```

Of course, there is an extra cost involved in building the sets in [Example 3-14](#), but if either the `needles` or the `haystack` is already a set, the alternatives in [Example 3-14](#) may be cheaper than [Example 3-13](#).

Any one of the preceding examples are capable of searching 1,000 elements in a haystack of 10,000,000 items in about 0.3 milliseconds—that’s close to 0.3 microseconds per element.

Besides the extremely fast membership test (thanks to the underlying hash table), the `set` and `frozenset` built-in types provide a rich API to create new sets or, in the case of `set`, to change existing ones. We will discuss the operations shortly, but first a note about syntax.

Set Literals

The syntax of set literals—`{1}`, `{1, 2}`, etc.—looks exactly like the math notation, with one important exception: there’s no literal notation for the empty set, so we must remember to write `set()`.



Syntax Quirk

Don’t forget that to create an empty set, you should use the constructor without an argument: `set()`. If you write `{}`, you’re creating an empty dict—this hasn’t changed in Python 3.

In Python 3, the standard string representation of sets always uses the `{...}` notation, except for the empty set:

```
>>> s = {1}
>>> type(s)
<class 'set'>
>>> s
{1}
>>> s.pop()
1
>>> s
set()
```

Literal set syntax like `{1, 2, 3}` is both faster and more readable than calling the constructor (e.g., `set([1, 2, 3])`). The latter form is slower because, to evaluate it, Python has to look up the set name to fetch the constructor, then build a list, and finally pass it to the constructor. In contrast, to process a literal like `{1, 2, 3}`, Python runs a specialized `BUILD_SET` bytecode.¹⁰

There is no special syntax to represent `frozenset` literals—they must be created by calling the constructor. The standard string representation in Python 3 looks like a `frozenset` constructor call. Note the output in the console session:

```
>>> frozenset(range(10))
frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
```

Speaking of syntax, the idea of listcomps was adapted to build sets as well.

Set Comprehensions

Set comprehensions (*setcomps*) were added way back in Python 2.7, together with the dictcomps that we saw in “[dict Comprehensions](#)” on page 79. [Example 3-15](#) shows how.

Example 3-15. Build a set of Latin-1 characters that have the word “SIGN” in their Unicode names

```
>>> from unicodedata import name ❶
>>> {chr(i) for i in range(32, 256) if 'SIGN' in name(chr(i), '')} ❷
{'Š', '=', 'Ç', '#', '¤', '<', '¥', 'µ', '×', '$', '¶', '£', '©',
'ª', '+', '÷', '±', '>', '¬', '®', '%'}
```

- ❶ Import `name` function from `unicodedata` to obtain character names.
- ❷ Build set of characters with codes from 32 to 255 that have the word `'SIGN'` in their names.

The order of the output changes for each Python process, because of the salted hash mentioned in “[What Is Hashable](#)” on page 84.

Syntax matters aside, let’s now consider the behavior of sets.

¹⁰ This may be interesting, but is not super important. The speed up will happen only when a set literal is evaluated, and that happens at most once per Python process—when a module is initially compiled. If you’re curious, import the `dis` function from the `dis` module and use it to disassemble the bytecodes for a set literal—e.g., `dis('{1}')`—and a set call—`dis('set([1])')`

Practical Consequences of How Sets Work

The set and frozenset types are both implemented with a hash table. This has these effects:

- Set elements must be hashable objects. They must implement proper `__hash__` and `__eq__` methods as described in “What Is Hashable” on page 84.
- Membership testing is very efficient. A set may have millions of elements, but an element can be located directly by computing its hash code and deriving an index offset, with the possible overhead of a small number of tries to find a matching element or exhaust the search.
- Sets have a significant memory overhead, compared to a low-level array pointers to its elements—which would be more compact but also much slower to search beyond a handful of elements.
- Element ordering depends on insertion order, but not in a useful or reliable way. If two elements are different but have the same hash code, their position depends on which element is added first.
- Adding elements to a set may change the order of existing elements. That’s because the algorithm becomes less efficient if the hash table is more than two-thirds full, so Python may need to move and resize the table as it grows. When this happens, elements are reinserted and their relative ordering may change.

See “Internals of sets and dicts” at fluentpython.com for details.

Let’s now review the rich assortment of operations provided by sets.

Set Operations

Figure 3-2 gives an overview of the methods you can use on mutable and immutable sets. Many of them are special methods that overload operators, such as `&` and `>=`. Table 3-2 shows the math set operators that have corresponding operators or methods in Python. Note that some operators and methods perform in-place changes on the target set (e.g., `&=`, `difference_update`, etc.). Such operations make no sense in the ideal world of mathematical sets, and are not implemented in `frozenset`.



The infix operators in [Table 3-2](#) require that both operands be sets, but all other methods take one or more iterable arguments. For example, to produce the union of four collections, *a*, *b*, *c*, and *d*, you can call `a.union(b, c, d)`, where *a* must be a set, but *b*, *c*, and *d* can be iterables of any type that produce hashable items. If you need to create a new set with the union of four iterables, instead of updating an existing set, you can write `{*a, *b, *c, *d}` since Python 3.5 thanks to [PEP 448—Additional Unpacking Generalizations](#).

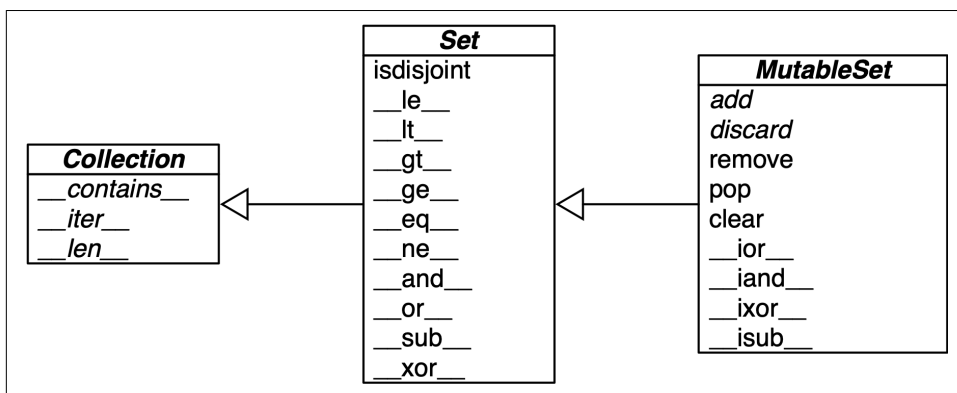


Figure 3-2. Simplified UML class diagram for `MutableSet` and its superclasses from `collections.abc` (names in *italic* are abstract classes and abstract methods; reverse operator methods omitted for brevity).

Table 3-2. Mathematical set operations: these methods either produce a new set or update the target set in place, if it's mutable

Math symbol	Python operator	Method	Description
$S \cap Z$	$s \ \& \ z$	<code>s.__and__(z)</code>	Intersection of <i>s</i> and <i>z</i>
	$z \ \& \ s$	<code>s.__rand__(z)</code>	Reversed <code>&</code> operator
		<code>s.intersection(it, ...)</code>	Intersection of <i>s</i> and all sets built from iterables <i>it</i> , etc.
	$s \ \&= \ z$	<code>s.__iand__(z)</code>	<i>s</i> updated with intersection of <i>s</i> and <i>z</i>
		<code>s.intersection_update(it, ...)</code>	<i>s</i> updated with intersection of <i>s</i> and all sets built from iterables <i>it</i> , etc.
$S \cup Z$	$s \ \ z$	<code>s.__or__(z)</code>	Union of <i>s</i> and <i>z</i>
	$z \ \ s$	<code>s.__ror__(z)</code>	Reversed <code> </code>
		<code>s.union(it, ...)</code>	Union of <i>s</i> and all sets built from iterables <i>it</i> , etc.

Math symbol	Python operator	Method	Description
$S \cup Z$	$s \mid= z$	<code>s.__ior__(z)</code>	<code>s</code> updated with union of <code>s</code> and <code>z</code>
		<code>s.update(it, ...)</code>	<code>s</code> updated with union of <code>s</code> and all sets built from iterables <code>it</code> , etc.
	$s - z$	<code>s.__sub__(z)</code>	Relative complement or difference between <code>s</code> and <code>z</code>
		<code>s.difference(it, ...)</code>	Reversed - operator Difference between <code>s</code> and all sets built from iterables <code>it</code> , etc.
$S \Delta Z$	$s \hat{=} z$	<code>s.__isub__(z)</code>	<code>s</code> updated with difference between <code>s</code> and <code>z</code>
		<code>s.difference_update(it, ...)</code>	<code>s</code> updated with difference between <code>s</code> and all sets built from iterables <code>it</code> , etc.
	$s \wedge z$	<code>s.__xor__(z)</code>	Symmetric difference (the complement of the intersection <code>s & z</code>)
		<code>s.__rxor__(z)</code>	Reversed \wedge operator
$S \Delta Z$	$s \wedge z$	<code>s.symmetric_difference(it)</code>	Complement of <code>s & set(it)</code>
		<code>s.__ixor__(z)</code>	<code>s</code> updated with symmetric difference of <code>s</code> and <code>z</code>
	$s \hat{=} z$	<code>s.symmetric_difference_update(it, ...)</code>	<code>s</code> updated with symmetric difference of <code>s</code> and all sets built from iterables <code>it</code> , etc.

Table 3-3 lists set predicates: operators and methods that return `True` or `False`.

Table 3-3. Set comparison operators and methods that return a bool

Math symbol	Python operator	Method	Description
$S \cap Z = \emptyset$		<code>s.isdisjoint(z)</code>	<code>s</code> and <code>z</code> are disjoint (no elements in common)
$e \in S$	<code>e in s</code>	<code>s.__contains__(e)</code>	Element <code>e</code> is a member of <code>s</code>
$S \subseteq Z$	<code>s <= z</code>	<code>s.__le__(z)</code>	<code>s</code> is a subset of the <code>z</code> set
		<code>s.issubset(it)</code>	<code>s</code> is a subset of the set built from the iterable <code>it</code>
$S \subset Z$	<code>s < z</code>	<code>s.__lt__(z)</code>	<code>s</code> is a proper subset of the <code>z</code> set
$S \supseteq Z$	<code>s >= z</code>	<code>s.__ge__(z)</code>	<code>s</code> is a superset of the <code>z</code> set
		<code>s.issuperset(it)</code>	<code>s</code> is a superset of the set built from the iterable <code>it</code>
$S \supset Z$	<code>s > z</code>	<code>s.__gt__(z)</code>	<code>s</code> is a proper superset of the <code>z</code> set

In addition to the operators and methods derived from math set theory, the set types implement other methods of practical use, summarized in **Table 3-4**.

Table 3-4. Additional set methods

	set	frozenset	
<code>s.add(e)</code>	•		Add element <code>e</code> to <code>s</code>
<code>s.clear()</code>	•		Remove all elements of <code>s</code>
<code>s.copy()</code>	•	•	Shallow copy of <code>s</code>
<code>s.discard(e)</code>	•		Remove element <code>e</code> from <code>s</code> if it is present
<code>s.__iter__()</code>	•	•	Get iterator over <code>s</code>
<code>s.__len__()</code>	•	•	<code>len(s)</code>
<code>s.pop()</code>	•		Remove and return an element from <code>s</code> , raising <code>KeyError</code> if <code>s</code> is empty
<code>s.remove(e)</code>	•		Remove element <code>e</code> from <code>s</code> , raising <code>KeyError</code> if <code>e not in s</code>

This completes our overview of the features of sets. As promised in “[Dictionary Views](#)” on page 101, we’ll now see how two of the dictionary view types behave very much like a `frozenset`.

Set Operations on dict Views

Table 3-5 shows that the view objects returned by the `dict` methods `.keys()` and `.items()` are remarkably similar to `frozenset`.

Table 3-5. Methods implemented by `frozenset`, `dict_keys`, and `dict_items`

	frozenset	dict_keys	dict_items	Description
<code>s.__and__(z)</code>	•	•	•	<code>s & z</code> (intersection of <code>s</code> and <code>z</code>)
<code>s.__rand__(z)</code>	•	•	•	Reversed <code>&</code> operator
<code>s.__contains__(e)</code>	•	•	•	<code>e in s</code>
<code>s.copy()</code>	•			Shallow copy of <code>s</code>
<code>s.difference(it, ...)</code>	•			Difference between <code>s</code> and iterables <code>it</code> , etc.
<code>s.intersection(it, ...)</code>	•			Intersection of <code>s</code> and iterables <code>it</code> , etc.
<code>s.isdisjoint(z)</code>	•	•	•	<code>s</code> and <code>z</code> are disjoint (no elements in common)
<code>s.issubset(it)</code>	•			<code>s</code> is a subset of iterable <code>it</code>
<code>s.issuperset(it)</code>	•			<code>s</code> is a superset of iterable <code>it</code>
<code>s.__iter__()</code>	•	•	•	Get iterator over <code>s</code>
<code>s.__len__()</code>	•	•	•	<code>len(s)</code>
<code>s.__or__(z)</code>	•	•	•	<code>s z</code> (union of <code>s</code> and <code>z</code>)
<code>s.__ror__(z)</code>	•	•	•	Reversed <code> </code> operator
<code>s.__reversed__()</code>		•	•	Get iterator over <code>s</code> in reverse order
<code>s.__rsub__(z)</code>	•	•	•	Reversed <code>-</code> operator
<code>s.__sub__(z)</code>	•	•	•	<code>s - z</code> (difference between <code>s</code> and <code>z</code>)

	frozenset	dict_keys	dict_items	Description
<code>s.symmetric_difference(it)</code>	•			Complement of <code>s & set(it)</code>
<code>s.union(it, ...)</code>	•			Union of <code>s</code> and iterables <code>it</code> , etc.
<code>s.__xor__()</code>	•	•	•	<code>s ^ z</code> (symmetric difference of <code>s</code> and <code>z</code>)
<code>s.__rxor__()</code>	•	•	•	Reversed <code>^</code> operator

In particular, `dict_keys` and `dict_items` implement the special methods to support the powerful set operators `&` (intersection), `|` (union), `-` (difference), and `^` (symmetric difference).

For example, using `&` is easy to get the keys that appear in two dictionaries:

```
>>> d1 = dict(a=1, b=2, c=3, d=4)
>>> d2 = dict(b=20, d=40, e=50)
>>> d1.keys() & d2.keys()
{'b', 'd'}
```

Note that the return value of `&` is a `set`. Even better: the set operators in dictionary views are compatible with `set` instances. Check this out:

```
>>> s = {'a', 'e', 'i'}
>>> d1.keys() & s
{'a'}
>>> d1.keys() | s
{'a', 'c', 'b', 'd', 'i', 'e'}
```



A `dict_items` view only works as a set if all values in the `dict` are hashable. Attempting set operations on a `dict_items` view with an unhashable value raises `TypeError: unhashable type 'T'`, with `T` as the type of the offending value.

On the other hand, a `dict_keys` view can always be used as a set, because every key is hashable—by definition.

Using set operators with views will save a lot of loops and ifs when inspecting the contents of dictionaries in your code. Let Python's efficient implementation in C work for you!

With this, we can wrap up this chapter.

Chapter Summary

Dictionaries are a keystone of Python. Over the years, the familiar `{k1: v1, k2: v2}` literal syntax was enhanced to support unpacking with `**`, pattern matching, as well as dict comprehensions.

Beyond the basic `dict`, the standard library offers handy, ready-to-use specialized mappings like `defaultdict`, `ChainMap`, and `Counter`, all defined in the `collections` module. With the new dict implementation, `OrderedDict` is not as useful as before, but should remain in the standard library for backward compatibility—and has specific characteristics that `dict` doesn't have, such as taking into account key ordering in `==` comparisons. Also in the `collections` module is the `UserDict`, an easy to use base class to create custom mappings.

Two powerful methods available in most mappings are `setdefault` and `update`. The `setdefault` method can update items holding mutable values—for example, in a dict of list values—avoiding a second search for the same key. The `update` method allows bulk insertion or overwriting of items from any other mapping, from iterables providing (key, value) pairs, and from keyword arguments. Mapping constructors also use `update` internally, allowing instances to be initialized from mappings, iterables, or keyword arguments. Since Python 3.9, we can also use the `|=` operator to update a mapping, and the `|` operator to create a new one from the union of two mappings.

A clever hook in the mapping API is the `__missing__` method, which lets you customize what happens when a key is not found when using the `d[k]` syntax that invokes `__getitem__`.

The `collections.abc` module provides the `Mapping` and `MutableMapping` abstract base classes as standard interfaces, useful for runtime type checking. The `MappingProxyType` from the `types` module creates an immutable façade for a mapping you want to protect from accidental change. There are also ABCs for `Set` and `MutableSet`.

Dictionary views were a great addition in Python 3, eliminating the memory overhead of the Python 2 `.keys()`, `.values()`, and `.items()` methods that built lists duplicating data in the target dict instance. In addition, the `dict_keys` and `dict_items` classes support the most useful operators and methods of `frozenset`.

Further Reading

In The Python Standard Library documentation, “[collections—Container datatypes](#)”, includes examples and practical recipes with several mapping types. The Python source code for the module *Lib/collections/__init__.py* is a great reference for anyone who wants to create a new mapping type or grok the logic of the existing ones. Chapter 1 of the *Python Cookbook, 3rd ed.* (O’Reilly) by David Beazley and Brian K. Jones has 20 handy and insightful recipes with data structures—the majority using dict in clever ways.

Greg Ganderberger advocates for the continued use of `collections.OrderedDict`, on the grounds that “explicit is better than implicit,” backward compatibility, and the fact that some tools and libraries assume the ordering of dict keys is irrelevant—his post: “[Python Dictionaries Are Now Ordered. Keep Using OrderedDict](#)”.

PEP 3106—[Revamping dict.keys\(\), .values\(\) and .items\(\)](#) is where Guido van Rossum presented the dictionary views feature for Python 3. In the abstract, he wrote that the idea came from the Java Collections Framework.

PyPy was the first Python interpreter to implement Raymond Hettinger’s proposal of compact dicts, and they blogged about it in “[Faster, more memory efficient and more ordered dictionaries on PyPy](#)”, acknowledging that a similar layout was adopted in PHP 7, described in [PHP’s new hashtable implementation](#). It’s always great when creators cite prior art.

At PyCon 2017, Brandon Rhodes presented “[The Dictionary Even Mightier](#)”, a sequel to his classic animated presentation “[The Mighty Dictionary](#)”—including animated hash collisions! Another up-to-date, but more in-depth video on the internals of Python’s dict is “[Modern Dictionaries](#)” by Raymond Hettinger, where he tells that after initially failing to sell compact dicts to the CPython core devs, he lobbied the PyPy team, they adopted it, the idea gained traction, and was finally [contributed](#) to CPython 3.6 by INADA Naoki. For all details, check out the extensive comments in the CPython code for *Objects/dictobject.c* and the design document *Objects/dict-notes.txt*.

The rationale for adding sets to Python is documented in [PEP 218—Adding a Built-In Set Object Type](#). When PEP 218 was approved, no special literal syntax was adopted for sets. The set literals were created for Python 3 and backported to Python 2.7, along with dict and set comprehensions. At PyCon 2019, I presented “[Set Practice: learning from Python’s set types](#)” describing use cases of sets in real programs, covering their API design, and the implementation of `uintset`, a set class for integer elements using a bit vector instead of a hash table, inspired by an example in Chapter 6 of the excellent *The Go Programming Language*, by Alan Donovan and Brian Kernighan (Addison-Wesley).

IEEE's *Spectrum* magazine has a story about Hans Peter Luhn, a prolific inventor who patented a punched card deck to select cocktail recipes depending on ingredients available, among other diverse inventions including...hash tables! See "[Hans Peter Luhn and the Birth of the Hashing Algorithm](#)".

Soapbox

Syntactic Sugar

My friend Geraldo Cohen once remarked that Python is "simple and correct."

Programming language purists like to dismiss syntax as unimportant.

Syntactic sugar causes cancer of the semicolon.

—Alan Perlis

Syntax is the user interface of a programming language, so it does matter in practice.

Before finding Python, I did some web programming using Perl and PHP. The syntax for mappings in these languages is very useful, and I badly miss it whenever I have to use Java or C.

A good literal syntax for mappings is very convenient for configuration, table-driven implementations, and to hold data for prototyping and testing. That's one lesson the designers of Go learned from dynamic languages. The lack of a good way to express structured data in code pushed the Java community to adopt the verbose and overly complex XML as a data format.

JSON was proposed as "[The Fat-Free Alternative to XML](#)" and became a huge success, replacing XML in many contexts. A concise syntax for lists and dictionaries makes an excellent data interchange format.

PHP and Ruby imitated the hash syntax from Perl, using `=>` to link keys to values. JavaScript uses `:` like Python. Why use two characters when one is readable enough?

JSON came from JavaScript, but it also happens to be an almost exact subset of Python syntax. JSON is compatible with Python except for the spelling of the values `true`, `false`, and `null`.

Armin Ronacher [tweeted](#) that he likes to hack Python's global namespace to add JSON-compatible aliases for Python's `True`, `False`, and `None` so he can paste JSON directly in the console. The basic idea:


```
>>> true, false, null = True, False, None
>>> fruit = {
...     "type": "banana",
...     "avg_weight": 123.2,
...     "edible_peel": false,
...     "species": ["acuminata", "balbisiana", "paradisiaca"],
...     "issues": null,
... }
>>> fruit
{'type': 'banana', 'avg_weight': 123.2, 'edible_peel': False,
 'species': ['acuminata', 'balbisiana', 'paradisiaca'], 'issues': None}
```

The syntax everybody now uses for exchanging data is Python's dict and list syntax. Now we have the nice syntax with the convenience of preserved insertion order.

Simple and correct.

Unicode Text Versus Bytes

Humans use text. Computers speak bytes.

—Esther Nam and Travis Fischer, “Character Encoding and Unicode in Python”¹

Python 3 introduced a sharp distinction between strings of human text and sequences of raw bytes. Implicit conversion of byte sequences to Unicode text is a thing of the past. This chapter deals with Unicode strings, binary sequences, and the encodings used to convert between them.

Depending on the kind of work you do with Python, you may think that understanding Unicode is not important. That’s unlikely, but anyway there is no escaping the `str` versus byte divide. As a bonus, you’ll find that the specialized binary sequence types provide features that the “all-purpose” Python 2 `str` type did not have.

In this chapter, we will visit the following topics:

- Characters, code points, and byte representations
- Unique features of binary sequences: `bytes`, `bytearray`, and `memoryview`
- Encodings for full Unicode and legacy character sets
- Avoiding and dealing with encoding errors
- Best practices when handling text files
- The default encoding trap and standard I/O issues
- Safe Unicode text comparisons with normalization

¹ Slide 12 of PyCon 2014 talk “Character Encoding and Unicode in Python” ([slides](#), [video](#)).

- Utility functions for normalization, case folding, and brute-force diacritic removal
- Proper sorting of Unicode text with `locale` and the *pyuca* library
- Character metadata in the Unicode database
- Dual-mode APIs that handle `str` and `bytes`

What's New in This Chapter

Support for Unicode in Python 3 has been comprehensive and stable, so the most notable addition is “[Finding Characters by Name](#)” on [page 151](#), describing a utility for searching the Unicode database—a great way to find circled digits and smiling cats from the command line.

One minor change worth mentioning is the Unicode support on Windows, which is better and simpler since Python 3.6, as we’ll see in “[Beware of Encoding Defaults](#)” on [page 134](#).

Let’s start with the not-so-new, but fundamental concepts of characters, code points, and bytes.



For the second edition, I expanded the section about the `struct` module and published it online at “[Parsing binary records with struct](#)”, in the *[fluentpython.com](#)* companion website.

There you will also find “[Building Multi-character Emojis](#)”, describing how to make country flags, rainbow flags, people with different skin tones, and diverse family icons by combining Unicode characters.

Character Issues

The concept of “string” is simple enough: a string is a sequence of characters. The problem lies in the definition of “character.”

In 2021, the best definition of “character” we have is a Unicode character. Accordingly, the items we get out of a Python 3 `str` are Unicode characters, just like the items of a `unicode` object in Python 2—and not the raw bytes we got from a Python 2 `str`.

The Unicode standard explicitly separates the identity of characters from specific byte representations:

- The identity of a character—its *code point*—is a number from 0 to 1,114,111 (base 10), shown in the Unicode standard as 4 to 6 hex digits with a “U+” prefix, from U+0000 to U+10FFFF. For example, the code point for the letter A is U+0041, the Euro sign is U+20AC, and the musical symbol G clef is assigned to code point U+1D11E. About 13% of the valid code points have characters assigned to them in Unicode 13.0.0, the standard used in Python 3.10.0b4.
- The actual bytes that represent a character depend on the *encoding* in use. An encoding is an algorithm that converts code points to byte sequences and vice versa. The code point for the letter A (U+0041) is encoded as the single byte `\x41` in the UTF-8 encoding, or as the bytes `\x41\x00` in UTF-16LE encoding. As another example, UTF-8 requires three bytes—`\xe2\x82\xac`—to encode the Euro sign (U+20AC), but in UTF-16LE the same code point is encoded as two bytes: `\xac\x20`.

Converting from code points to bytes is *encoding*; converting from bytes to code points is *decoding*. See [Example 4-1](#).

Example 4-1. Encoding and decoding

```
>>> s = 'café'
>>> len(s) ❶
4
>>> b = s.encode('utf8') ❷
>>> b
b'caf\xc3\xa9' ❸
>>> len(b) ❹
5
>>> b.decode('utf8') ❺
'café'
```

- ❶ The `str` `'café'` has four Unicode characters.
- ❷ Encode `str` to bytes using UTF-8 encoding.
- ❸ bytes literals have a `b` prefix.
- ❹ bytes `b` has five bytes (the code point for “é” is encoded as two bytes in UTF-8).
- ❺ Decode bytes to `str` using UTF-8 encoding.



If you need a memory aid to help distinguish `.decode()` from `.encode()`, convince yourself that byte sequences can be cryptic machine core dumps, while Unicode `str` objects are “human” text. Therefore, it makes sense that we *decode* bytes to `str` to get human-readable text, and we *encode* `str` to bytes for storage or transmission.

Although the Python 3 `str` is pretty much the Python 2 `unicode` type with a new name, the Python 3 `bytes` is not simply the old `str` renamed, and there is also the closely related `bytearray` type. So it is worthwhile to take a look at the binary sequence types before advancing to encoding/decoding issues.

Byte Essentials

The new binary sequence types are unlike the Python 2 `str` in many regards. The first thing to know is that there are two basic built-in types for binary sequences: the immutable `bytes` type introduced in Python 3 and the mutable `bytearray`, added way back in Python 2.6.² The Python documentation sometimes uses the generic term “byte string” to refer to both `bytes` and `bytearray`. I avoid that confusing term.

Each item in `bytes` or `bytearray` is an integer from 0 to 255, and not a one-character string like in the Python 2 `str`. However, a slice of a binary sequence always produces a binary sequence of the same type—including slices of length 1. See [Example 4-2](#).

Example 4-2. A five-byte sequence as `bytes` and as `bytearray`

```
>>> cafe = bytes('café', encoding='utf_8') ❶
>>> cafe
b'caf\xc3\xa9'
>>> cafe[0] ❷
99
>>> cafe[:1] ❸
b'c'
>>> cafe_arr = bytearray(cafe)
>>> cafe_arr ❹
bytearray(b'caf\xc3\xa9')
>>> cafe_arr[-1:] ❺
bytearray(b'\xa9')
```

- ❶ `bytes` can be built from a `str`, given an encoding.
- ❷ Each item is an integer in `range(256)`.

² Python 2.6 and 2.7 also had `bytes`, but it was just an alias to the `str` type.

- ③ Slices of bytes are also bytes—even slices of a single byte.
- ④ There is no literal syntax for bytearray: they are shown as bytearray() with a bytes literal as argument.
- ⑤ A slice of bytearray is also a bytearray.



The fact that `my_bytes[0]` retrieves an `int` but `my_bytes[:1]` returns a bytes sequence of length 1 is only surprising because we are used to Python's `str` type, where `s[0] == s[:1]`. For all other sequence types in Python, 1 item is not the same as a slice of length 1.

Although binary sequences are really sequences of integers, their literal notation reflects the fact that ASCII text is often embedded in them. Therefore, four different displays are used, depending on each byte value:

- For bytes with decimal codes 32 to 126—from space to ~ (tilde)—the ASCII character itself is used.
- For bytes corresponding to tab, newline, carriage return, and \, the escape sequences `\t`, `\n`, `\r`, and `\\` are used.
- If both string delimiters ' and " appear in the byte sequence, the whole sequence is delimited by ', and any ' inside are escaped as \'.³
- For other byte values, a hexadecimal escape sequence is used (e.g., `\x00` is the null byte).

That is why in [Example 4-2](#) you see `b'caf\xc3\xa9'`: the first three bytes `b'caf'` are in the printable ASCII range, the last two are not.

Both bytes and bytearray support every `str` method except those that do formatting (`format`, `format_map`) and those that depend on Unicode data, including `case fold`, `isdecimal`, `isidentifier`, `isnumeric`, `isprintable`, and `encode`. This means that you can use familiar string methods like `endswith`, `replace`, `strip`, `translate`, `upper`, and dozens of others with binary sequences—only using bytes and not `str` arguments. In addition, the regular expression functions in the `re` module also work

³ Trivia: the ASCII “single quote” character that Python uses by default as the string delimiter is actually named APOSTROPHE in the Unicode standard. The real single quotes are asymmetric: left is U+2018 and right is U+2019.

on binary sequences, if the regex is compiled from a binary sequence instead of a `str`. Since Python 3.5, the `%` operator works with binary sequences again.⁴

Binary sequences have a class method that `str` doesn't have, called `fromhex`, which builds a binary sequence by parsing pairs of hex digits optionally separated by spaces:

```
>>> bytes.fromhex('31 4B CE A9')
b'1K\xce\xa9'
```

The other ways of building `bytes` or `bytearray` instances are calling their constructors with:

- A `str` and an encoding keyword argument
- An iterable providing items with values from 0 to 255
- An object that implements the buffer protocol (e.g., `bytes`, `bytearray`, `memory view`, `array.array`) that copies the bytes from the source object to the newly created binary sequence



Until Python 3.5, it was also possible to call `bytes` or `bytearray` with a single integer to create a binary sequence of that size initialized with null bytes. This signature was deprecated in Python 3.5 and removed in Python 3.6. See [PEP 467—Minor API improvements for binary sequences](#).

Building a binary sequence from a buffer-like object is a low-level operation that may involve type casting. See a demonstration in [Example 4-3](#).

Example 4-3. Initializing bytes from the raw data of an array

```
>>> import array
>>> numbers = array.array('h', [-2, -1, 0, 1, 2]) ❶
>>> octets = bytes(numbers) ❷
>>> octets
b'\xfe\xff\xff\xff\x00\x00\x01\x00\x02\x00' ❸
```

- ❶ Typecode `'h'` creates an array of short integers (16 bits).
- ❷ `octets` holds a copy of the bytes that make up `numbers`.
- ❸ These are the 10 bytes that represent the 5 short integers.

⁴ It did not work in Python 3.0 to 3.4, causing much pain to developers dealing with binary data. The reversal is documented in [PEP 461—Adding % formatting to bytes and bytearray](#).

Creating a `bytes` or `bytearray` object from any buffer-like source will always copy the bytes. In contrast, `memoryview` objects let you share memory between binary data structures, as we saw in “Memory Views” on page 62.

After this basic exploration of binary sequence types in Python, let’s see how they are converted to/from strings.

Basic Encoders/Decoders

The Python distribution bundles more than 100 *codecs* (encoder/decoders) for text to byte conversion and vice versa. Each codec has a name, like `'utf_8'`, and often aliases, such as `'utf8'`, `'utf-8'`, and `'U8'`, which you can use as the encoding argument in functions like `open()`, `str.encode()`, `bytes.decode()`, and so on.

Example 4-4 shows the same text encoded as three different byte sequences.

Example 4-4. The string “El Niño” encoded with three codecs producing very different byte sequences

```
>>> for codec in ['latin_1', 'utf_8', 'utf_16']:
...     print(codec, 'El Niño'.encode(codec), sep='\t')
...
latin_1 b'El Ni\xf1o'
utf_8   b'El Ni\xc3\xb1o'
utf_16  b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xfa\x00o\x00'
```

Figure 4-1 demonstrates a variety of codecs generating bytes from characters like the letter “A” through the G-clef musical symbol. Note that the last three encodings are variable-length, multibyte encodings.

char.	code point	ascii	latin1	cp1252	cp437	gb2312	utf-8	utf-16le
A	U+0041	41	41	41	41	41	41	41 00
¿	U+00BF	*	BF	BF	A8	*	C2 BF	BF 00
Ã	U+00C3	*	C3	C3	*	*	C3 83	C3 00
á	U+00E1	*	E1	E1	A0	A8 A2	C3 A1	E1 00
Ω	U+03A9	*	*	*	EA	A6 B8	CE A9	A9 03
€	U+06BF	*	*	*	*	*	DA BF	BF 06
“	U+201C	*	*	93	*	A1 B0	E2 80 9C	1C 20
€	U+20AC	*	*	80	*	*	E2 82 AC	AC 20
Г	U+250C	*	*	*	DA	A9 B0	E2 94 8C	0C 25
气	U+6C14	*	*	*	*	C6 F8	E6 B0 94	14 6C
氣	U+6C23	*	*	*	*	*	E6 B0 A3	23 6C
𝄞	U+1D11E	*	*	*	*	*	F0 9D 84 9E	34 D8 1E DD

Figure 4-1. Twelve characters, their code points, and their byte representation (in hex) in 7 different encodings (asterisks indicate that the character cannot be represented in that encoding).

All those asterisks in [Figure 4-1](#) make clear that some encodings, like ASCII and even the multibyte GB2312, cannot represent every Unicode character. The UTF encodings, however, are designed to handle every Unicode code point.

The encodings shown in [Figure 4-1](#) were chosen as a representative sample:

`latin1 a.k.a. iso8859_1`

Important because it is the basis for other encodings, such as `cp1252` and Unicode itself (note how the `latin1` byte values appear in the `cp1252` bytes and even in the code points).

`cp1252`

A useful `latin1` superset created by Microsoft, adding useful symbols like curly quotes and € (euro); some Windows apps call it “ANSI,” but it was never a real ANSI standard.

`cp437`

The original character set of the IBM PC, with box drawing characters. Incompatible with `latin1`, which appeared later.

`gb2312`

Legacy standard to encode the simplified Chinese ideographs used in mainland China; one of several widely deployed multibyte encodings for Asian languages.

`utf-8`

The most common 8-bit encoding on the web, by far, as of July 2021, “[W³Techs: Usage statistics of character encodings for websites](#)” claims that 97% of sites use UTF-8, up from 81.4% when I wrote this paragraph in the first edition of this book in September 2014.

`utf-16le`

One form of the UTF 16-bit encoding scheme; all UTF-16 encodings support code points beyond U+FFFF through escape sequences called “surrogate pairs.”



UTF-16 superseded the original 16-bit Unicode 1.0 encoding—UCS-2—way back in 1996. UCS-2 is still used in many systems despite being deprecated since the last century because it only supports code points up to U+FFFF. As of 2021, more than 57% of the allocated code points are above U+FFFF, including the all-important emojis.

With this overview of common encodings now complete, we move to handling issues in encoding and decoding operations.

Understanding Encode/Decode Problems

Although there is a generic `UnicodeError` exception, the error reported by Python is usually more specific: either a `UnicodeEncodeError` (when converting `str` to binary sequences) or a `UnicodeDecodeError` (when reading binary sequences into `str`). Loading Python modules may also raise `SyntaxError` when the source encoding is unexpected. We'll show how to handle all of these errors in the next sections.



The first thing to note when you get a Unicode error is the exact type of the exception. Is it a `UnicodeEncodeError`, a `UnicodeDecodeError`, or some other error (e.g., `SyntaxError`) that mentions an encoding problem? To solve the problem, you have to understand it first.

Coping with UnicodeEncodeError

Most non-UTF codecs handle only a small subset of the Unicode characters. When converting text to bytes, if a character is not defined in the target encoding, `UnicodeEncodeError` will be raised, unless special handling is provided by passing an `errors` argument to the encoding method or function. The behavior of the error handlers is shown in [Example 4-5](#).

Example 4-5. Encoding to bytes: success and error handling

```
>>> city = 'São Paulo'
>>> city.encode('utf_8') ❶
b'S\xc3\xa3o Paulo'
>>> city.encode('utf_16')
b'\xff\xfeS\x00\xe3\x00o\x00 \x00P\x00a\x00u\x00l\x00o\x00'
>>> city.encode('iso8859_1') ❷
b'S\xe3o Paulo'
>>> city.encode('cp437') ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "../lib/python3.4/encodings/cp437.py", line 12, in encode
    return codecs.charmap_encode(input,errors,encoding_map)
UnicodeEncodeError: 'charmap' codec can't encode character '\xe3' in
position 1: character maps to <undefined>
>>> city.encode('cp437', errors='ignore') ❹
b'So Paulo'
>>> city.encode('cp437', errors='replace') ❺
b'S?o Paulo'
>>> city.encode('cp437', errors='xmlcharrefreplace') ❻
b'S&#227;o Paulo'
```

- ❶ The UTF encodings handle any `str`.
- ❷ `iso8859_1` also works for the `'São Paulo'` string.
- ❸ `cp437` can't encode the `'ã'` (“a” with tilde). The default error handler — `'strict'` — raises `UnicodeEncodeError`.
- ❹ The `error='ignore'` handler skips characters that cannot be encoded; this is usually a very bad idea, leading to silent data loss.
- ❺ When encoding, `error='replace'` substitutes unencodable characters with `'?'`; data is also lost, but users will get a clue that something is amiss.
- ❻ `'xmlcharrefreplace'` replaces unencodable characters with an XML entity. If you can't use UTF, and you can't afford to lose data, this is the only option.



The codecs error handling is extensible. You may register extra strings for the `errors` argument by passing a name and an error handling function to the `codecs.register_error` function. See [the `codecs.register_error` documentation](#).

ASCII is a common subset to all the encodings that I know about, therefore encoding should always work if the text is made exclusively of ASCII characters. Python 3.7 added a new boolean method `str.isascii()` to check whether your Unicode text is 100% pure ASCII. If it is, you should be able to encode it to bytes in any encoding without raising `UnicodeEncodeError`.

Coping with UnicodeDecodeError

Not every byte holds a valid ASCII character, and not every byte sequence is valid UTF-8 or UTF-16; therefore, when you assume one of these encodings while converting a binary sequence to text, you will get a `UnicodeDecodeError` if unexpected bytes are found.

On the other hand, many legacy 8-bit encodings like `'cp1252'`, `'iso8859_1'`, and `'koi8_r'` are able to decode any stream of bytes, including random noise, without reporting errors. Therefore, if your program assumes the wrong 8-bit encoding, it will silently decode garbage.



Garbled characters are known as gremlins or mojibake (文字化け —Japanese for “transformed text”).

Example 4-6 illustrates how using the wrong codec may produce gremlins or a `UnicodeDecodeError`.

Example 4-6. Decoding from `str` to bytes: success and error handling

```
>>> octets = b'Montr\xe9al' ❶
>>> octets.decode('cp1252') ❷
'Montréal'
>>> octets.decode('iso8859_7') ❸
'Montrial'
>>> octets.decode('koi8_r') ❹
'MontrMal'
>>> octets.decode('utf_8') ❺
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position 5:
invalid continuation byte
>>> octets.decode('utf_8', errors='replace') ❻
'Montr  al'
```

- ❶ The word “Montréal” encoded as `latin1`; `'\xe9'` is the byte for “é”.
- ❷ Decoding with Windows 1252 works because it is a superset of `latin1`.
- ❸ ISO-8859-7 is intended for Greek, so the `'\xe9'` byte is misinterpreted, and no error is issued.
- ❹ KOI8-R is for Russian. Now `'\xe9'` stands for the Cyrillic letter “И”.
- ❺ The `'utf_8'` codec detects that `octets` is not valid UTF-8, and raises `UnicodeDecodeError`.
- ❻ Using `'replace'` error handling, the `\xe9` is replaced by “  ” (code point U+FFFD), the official Unicode REPLACEMENT CHARACTER intended to represent unknown characters.

SyntaxError When Loading Modules with Unexpected Encoding

UTF-8 is the default source encoding for Python 3, just as ASCII was the default for Python 2. If you load a `.py` module containing non-UTF-8 data and no encoding declaration, you get a message like this:

```
SyntaxError: Non-UTF-8 code starting with '\xe1' in file ola.py on line
1, but no encoding declared; see https://python.org/dev/peps/pep-0263/
for details
```

Because UTF-8 is widely deployed in GNU/Linux and macOS systems, a likely scenario is opening a `.py` file created on Windows with `cp1252`. Note that this error happens even in Python for Windows, because the default encoding for Python 3 source is UTF-8 across all platforms.

To fix this problem, add a magic coding comment at the top of the file, as shown in [Example 4-7](#).

Example 4-7. ola.py: “Hello, World!” in Portuguese

```
# coding: cp1252

print('Olá, Mundo!')
```



Now that Python 3 source code is no longer limited to ASCII and defaults to the excellent UTF-8 encoding, the best “fix” for source code in legacy encodings like `'cp1252'` is to convert them to UTF-8 already, and not bother with the coding comments. If your editor does not support UTF-8, it's time to switch.

Suppose you have a text file, be it source code or poetry, but you don't know its encoding. How do you detect the actual encoding? Answers in the next section.

How to Discover the Encoding of a Byte Sequence

How do you find the encoding of a byte sequence? Short answer: you can't. You must be told.

Some communication protocols and file formats, like HTTP and XML, contain headers that explicitly tell us how the content is encoded. You can be sure that some byte streams are not ASCII because they contain byte values over 127, and the way UTF-8 and UTF-16 are built also limits the possible byte sequences.

Leo's Hack for Guessing UTF-8 Decoding

(The next paragraphs come from a note left by tech reviewer Leonardo Rochael in the draft of this book.)

The way UTF-8 was designed, it's almost impossible for a random sequence of bytes, or even a nonrandom sequence of bytes coming from a non-UTF-8 encoding, to be decoded accidentally as garbage in UTF-8, instead of raising `UnicodeDecodeError`.

The reasons for this are that UTF-8 escape sequences never use ASCII characters, and these escape sequences have bit patterns that make it very hard for random data to be valid UTF-8 by accident.

So if you can decode some bytes containing codes > 127 as UTF-8, it's probably UTF-8.

In dealing with Brazilian online services, some of which were attached to legacy back-ends, I've had, on occasion, to implement a decoding strategy of trying to decode via UTF-8 and treat a `UnicodeDecodeError` by decoding via `cp1252`. It was ugly but effective.

However, considering that human languages also have their rules and restrictions, once you assume that a stream of bytes is human *plain text*, it may be possible to sniff out its encoding using heuristics and statistics. For example, if `b'\x00'` bytes are common, it is probably a 16- or 32-bit encoding, and not an 8-bit scheme, because null characters in plain text are bugs. When the byte sequence `b'\x20\x00'` appears often, it is more likely to be the space character (U+0020) in a UTF-16LE encoding, rather than the obscure U+2000 EN QUAD character—whatever that is.

That is how the package “**Chardet—The Universal Character Encoding Detector**” works to guess one of more than 30 supported encodings. *Chardet* is a Python library that you can use in your programs, but also includes a command-line utility, `chardetect`. Here is what it reports on the source file for this chapter:

```
$ chardetect 04-text-byte.asciidoc
04-text-byte.asciidoc: utf-8 with confidence 0.99
```

Although binary sequences of encoded text usually don't carry explicit hints of their encoding, the UTF formats may prepend a byte order mark to the textual content. That is explained next.

BOM: A Useful Gremlin

In **Example 4-4**, you may have noticed a couple of extra bytes at the beginning of a UTF-16 encoded sequence. Here they are again:

```
>>> u16 = 'El Niño'.encode('utf_16')
>>> u16
b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xfa\x00o\x00'
```

The bytes are `b'\xff\xfe'`. That is a *BOM*—byte-order mark—denoting the “little-endian” byte ordering of the Intel CPU where the encoding was performed.

On a little-endian machine, for each code point the least significant byte comes first: the letter 'E', code point U+0045 (decimal 69), is encoded in byte offsets 2 and 3 as 69 and 0:

```
>>> list(u16)
[255, 254, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
```

On a big-endian CPU, the encoding would be reversed; 'E' would be encoded as 0 and 69.

To avoid confusion, the UTF-16 encoding prepends the text to be encoded with the special invisible character ZERO WIDTH NO-BREAK SPACE (U+FEFF). On a little-endian system, that is encoded as `b'\xff\xfe'` (decimal 255, 254). Because, by design, there is no U+FFFE character in Unicode, the byte sequence `b'\xff\xfe'` must mean the ZERO WIDTH NO-BREAK SPACE on a little-endian encoding, so the codec knows which byte ordering to use.

There is a variant of UTF-16—UTF-16LE—that is explicitly little-endian, and another one explicitly big-endian, UTF-16BE. If you use them, a BOM is not generated:

```
>>> u16le = 'El Niño'.encode('utf_16le')
>>> list(u16le)
[69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
>>> u16be = 'El Niño'.encode('utf_16be')
>>> list(u16be)
[0, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111]
```

If present, the BOM is supposed to be filtered by the UTF-16 codec, so that you only get the actual text contents of the file without the leading ZERO WIDTH NO-BREAK SPACE. The Unicode standard says that if a file is UTF-16 and has no BOM, it should be assumed to be UTF-16BE (big-endian). However, the Intel x86 architecture is little-endian, so there is plenty of little-endian UTF-16 with no BOM in the wild.

This whole issue of endianness only affects encodings that use words of more than one byte, like UTF-16 and UTF-32. One big advantage of UTF-8 is that it produces the same byte sequence regardless of machine endianness, so no BOM is needed. Nevertheless, some Windows applications (notably Notepad) add the BOM to UTF-8 files anyway—and Excel depends on the BOM to detect a UTF-8 file, otherwise it assumes the content is encoded with a Windows code page. This UTF-8 encoding with BOM is called UTF-8-SIG in Python’s codec registry. The character U+FEFF

encoded in UTF-8-SIG is the three-byte sequence `b'\xef\xbb\xbf'`. So if a file starts with those three bytes, it is likely to be a UTF-8 file with a BOM.



Caleb's Tip about UTF-8-SIG

Caleb Hattingh—one of the tech reviewers—suggests always using the UTF-8-SIG codec when reading UTF-8 files. This is harmless because UTF-8-SIG reads files with or without a BOM correctly, and does not return the BOM itself. When writing, I recommend using UTF-8 for general interoperability. For example, Python scripts can be made executable in Unix systems if they start with the comment: `#!/usr/bin/env python3`. The first two bytes of the file must be `b'#!'` for that to work, but the BOM breaks that convention. If you have a specific requirement to export data to apps that need the BOM, use UTF-8-SIG but be aware that Python's [codecs documentation](#) says: “In UTF-8, the use of the BOM is discouraged and should generally be avoided.”

We now move on to handling text files in Python 3.

Handling Text Files

The best practice for handling text I/O is the “Unicode sandwich” ([Figure 4-2](#)).⁵ This means that bytes should be decoded to `str` as early as possible on input (e.g., when opening a file for reading). The “filling” of the sandwich is the business logic of your program, where text handling is done exclusively on `str` objects. You should never be encoding or decoding in the middle of other processing. On output, the `str` are encoded to bytes as late as possible. Most web frameworks work like that, and we rarely touch bytes when using them. In Django, for example, your views should output Unicode `str`; Django itself takes care of encoding the response to bytes, using UTF-8 by default.

Python 3 makes it easier to follow the advice of the Unicode sandwich, because the `open()` built-in does the necessary decoding when reading and encoding when writing files in text mode, so all you get from `my_file.read()` and pass to `my_file.write(text)` are `str` objects.

Therefore, using text files is apparently simple. But if you rely on default encodings, you will get bitten.

⁵ I first saw the term “Unicode sandwich” in Ned Batchelder’s excellent “[Pragmatic Unicode](#)” talk at US PyCon 2012.

The Unicode sandwich



bytes → str

Decode bytes on input,

100% str

process text only,

str → bytes

encode text on output.

Figure 4-2. Unicode sandwich: current best practice for text processing.

Consider the console session in [Example 4-8](#). Can you spot the bug?

Example 4-8. A platform encoding issue (if you try this on your machine, you may or may not see the problem)

```
>>> open('cafe.txt', 'w', encoding='utf_8').write('café')
4
>>> open('cafe.txt').read()
'cafÃ©'
```

The bug: I specified UTF-8 encoding when writing the file but failed to do so when reading it, so Python assumed Windows default file encoding—code page 1252—and the trailing bytes in the file were decoded as characters 'Ã©' instead of 'é'.

I ran [Example 4-8](#) on Python 3.8.1, 64 bits, on Windows 10 (build 18363). The same statements running on recent GNU/Linux or macOS work perfectly well because their default encoding is UTF-8, giving the false impression that everything is fine. If the encoding argument was omitted when opening the file to write, the locale default encoding would be used, and we'd read the file correctly using the same encoding. But then this script would generate files with different byte contents depending on the platform or even depending on locale settings in the same platform, creating compatibility problems.



Code that has to run on multiple machines or on multiple occasions should never depend on encoding defaults. Always pass an explicit `encoding=` argument when opening text files, because the default may change from one machine to the next, or from one day to the next.

A curious detail in [Example 4-8](#) is that the write function in the first statement reports that four characters were written, but in the next line five characters are read. [Example 4-9](#) is an extended version of [Example 4-8](#), explaining that and other details.

Example 4-9. Closer inspection of [Example 4-8](#) running on Windows reveals the bug and how to fix it

```
>>> fp = open('cafe.txt', 'w', encoding='utf_8')
>>> fp ❶
<_io.TextIOWrapper name='cafe.txt' mode='w' encoding='utf_8'>
>>> fp.write('café') ❷
4
>>> fp.close()
>>> import os
>>> os.stat('cafe.txt').st_size ❸
5
>>> fp2 = open('cafe.txt')
>>> fp2 ❹
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='cp1252'>
>>> fp2.encoding ❺
'cp1252'
>>> fp2.read() ❻
'cafÃ©'
>>> fp3 = open('cafe.txt', encoding='utf_8') ❼
>>> fp3
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='utf_8'>
>>> fp3.read() ❽
'café'
>>> fp4 = open('cafe.txt', 'rb') ❾
>>> fp4
<_io.BufferedReader name='cafe.txt'>
>>> fp4.read() ❿
b'caf\xc3\xa9'
```

- ❶ By default, open uses text mode and returns a TextIOWrapper object with a specific encoding.
- ❷ The write method on a TextIOWrapper returns the number of Unicode characters written.
- ❸ os.stat says the file has 5 bytes; UTF-8 encodes 'é' as 2 bytes, 0xc3 and 0xa9.
- ❹ Opening a text file with no explicit encoding returns a TextIOWrapper with the encoding set to a default from the locale.
- ❺ A TextIOWrapper object has an encoding attribute that you can inspect: cp1252 in this case.

- ❹ In the Windows cp1252 encoding, the byte 0xc3 is an “Ã” (A with tilde), and 0xa9 is the copyright sign.
- ❺ Opening the same file with the correct encoding.
- ❻ The expected result: the same four Unicode characters for 'café'.
- ❼ The 'rb' flag opens a file for reading in binary mode.
- ❽ The returned object is a `BufferedReader` and not a `TextIOWrapper`.
- ❾ Reading that returns bytes, as expected.



Do not open text files in binary mode unless you need to analyze the file contents to determine the encoding—even then, you should be using Chardet instead of reinventing the wheel (see “[How to Discover the Encoding of a Byte Sequence](#)” on page 128). Ordinary code should only use binary mode to open binary files, like raster images.

The problem in [Example 4-9](#) has to do with relying on a default setting while opening a text file. There are several sources for such defaults, as the next section shows.

Beware of Encoding Defaults

Several settings affect the encoding defaults for I/O in Python. See the `default_encodings.py` script in [Example 4-10](#).

Example 4-10. Exploring encoding defaults

```
import locale
import sys

expressions = """
    locale.getpreferredencoding()
    type(my_file)
    my_file.encoding
    sys.stdout.isatty()
    sys.stdout.encoding
    sys.stdin.isatty()
    sys.stdin.encoding
    sys.stderr.isatty()
    sys.stderr.encoding
    sys.getdefaultencoding()
    sys.getfilesystemencoding()
    """
```

```

my_file = open('dummy', 'w')

for expression in expressions.split():
    value = eval(expression)
    print(f'{expression:>30} -> {value!r}')

```

The output of [Example 4-10](#) on GNU/Linux (Ubuntu 14.04 to 19.10) and macOS (10.9 to 10.14) is identical, showing that UTF-8 is used everywhere in these systems:

```

$ python3 default_encodings.py
locale.getpreferredencoding() -> 'UTF-8'
type(my_file) -> <class '_io.TextIOWrapper'>
my_file.encoding -> 'UTF-8'
sys.stdout.isatty() -> True
sys.stdout.encoding -> 'utf-8'
sys.stdin.isatty() -> True
sys.stdin.encoding -> 'utf-8'
sys.stderr.isatty() -> True
sys.stderr.encoding -> 'utf-8'
sys.getdefaultencoding() -> 'utf-8'
sys.getfilesystemencoding() -> 'utf-8'

```

On Windows, however, the output is [Example 4-11](#).

Example 4-11. Default encodings on Windows 10 PowerShell (output is the same on cmd.exe)

```

> chcp ❶
Active code page: 437
> python default_encodings.py ❷
locale.getpreferredencoding() -> 'cp1252' ❸
type(my_file) -> <class '_io.TextIOWrapper'>
my_file.encoding -> 'cp1252' ❹
sys.stdout.isatty() -> True ❺
sys.stdout.encoding -> 'utf-8' ❻
sys.stdin.isatty() -> True
sys.stdin.encoding -> 'utf-8'
sys.stderr.isatty() -> True
sys.stderr.encoding -> 'utf-8'
sys.getdefaultencoding() -> 'utf-8'
sys.getfilesystemencoding() -> 'utf-8'

```

- ❶ chcp shows the active code page for the console: 437.
- ❷ Running `default_encodings.py` with output to console.
- ❸ `locale.getpreferredencoding()` is the most important setting.
- ❹ Text files use `locale.getpreferredencoding()` by default.

- 5 The output is going to the console, so `sys.stdout.isatty()` is `True`.
- 6 Now, `sys.stdout.encoding` is not the same as the console code page reported by `chcp`!

Unicode support in Windows itself, and in Python for Windows, got better since I wrote the first edition of this book. [Example 4-11](#) used to report four different encodings in Python 3.4 on Windows 7. The encodings for `stdout`, `stdin`, and `stderr` used to be the same as the active code page reported by the `chcp` command, but now they're all `utf-8` thanks to [PEP 528—Change Windows console encoding to UTF-8](#) implemented in Python 3.6, and Unicode support in PowerShell in *cmd.exe* (since Windows 1809 from October 2018).⁶ It's weird that `chcp` and `sys.stdout.encoding` say different things when `stdout` is writing to the console, but it's great that now we can print Unicode strings without encoding errors on Windows—unless the user redirects output to a file, as we'll soon see. That does not mean all your favorite emojis will appear in the console: that also depends on the font the console is using.

Another change was [PEP 529—Change Windows filesystem encoding to UTF-8](#), also implemented in Python 3.6, which changed the filesystem encoding (used to represent names of directories and files) from Microsoft's proprietary MBCS to UTF-8.

However, if the output of [Example 4-10](#) is redirected to a file, like this:

```
Z:\>python default_encodings.py > encodings.log
```

then, the value of `sys.stdout.isatty()` becomes `False`, and `sys.stdout.encoding` is set by `locale.getpreferredencoding()`, `'cp1252'` in that machine—but `sys.stdin.encoding` and `sys.stderr.encoding` remain `utf-8`.



In [Example 4-12](#) I use the `'\N{'` escape for Unicode literals, where we write the official name of the character inside the `\N{'`. It's rather verbose, but explicit and safe: Python raises `SyntaxError` if the name doesn't exist—much better than writing a hex number that could be wrong, but you'll only find out much later. You'd probably want to write a comment explaining the character codes anyway, so the verbosity of `\N{'` is easy to accept.

This means that a script like [Example 4-12](#) works when printing to the console, but may break when output is redirected to a file.

⁶ Source: "[Windows Command-Line: Unicode and UTF-8 Output Text Buffer](#)".

Example 4-12. stdout_check.py

```
import sys
from unicodedata import name

print(sys.version)
print()
print('sys.stdout.isatty():', sys.stdout.isatty())
print('sys.stdout.encoding:', sys.stdout.encoding)
print()

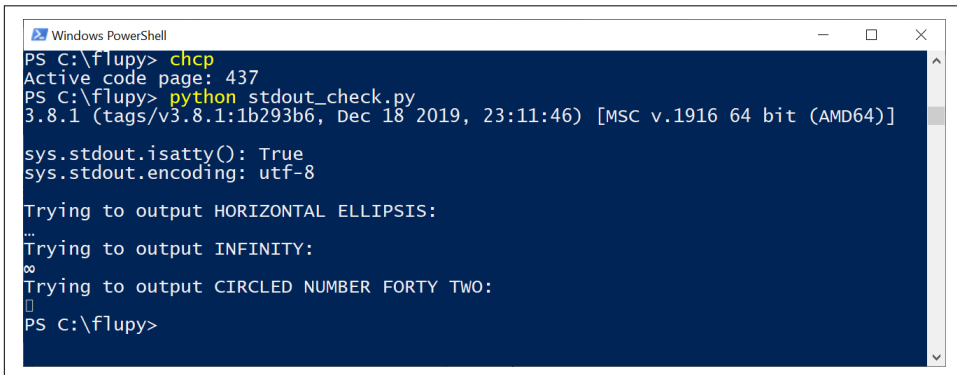
test_chars = [
    '\N{HORIZONTAL ELLIPSIS}',      # exists in cp1252, not in cp437
    '\N{INFINITY}',                  # exists in cp437, not in cp1252
    '\N{CIRCLED NUMBER FORTY TWO}', # not in cp437 or in cp1252
]

for char in test_chars:
    print(f'Trying to output {name(char)}:')
    print(char)
```

Example 4-12 displays the result of `sys.stdout.isatty()`, the value of `sys.stdout.encoding`, and these three characters:

- '...' HORIZONTAL ELLIPSIS—exists in CP 1252 but not in CP 437.
- '∞' INFINITY—exists in CP 437 but not in CP 1252.
- 'Ⓣ' CIRCLED NUMBER FORTY TWO—doesn't exist in CP 1252 or CP 437.

When I run `stdout_check.py` on PowerShell or `cmd.exe`, it works as captured in Figure 4-3.



```
Windows PowerShell
PS C:\flupy> chcp
Active code page: 437
PS C:\flupy> python stdout_check.py
3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)]

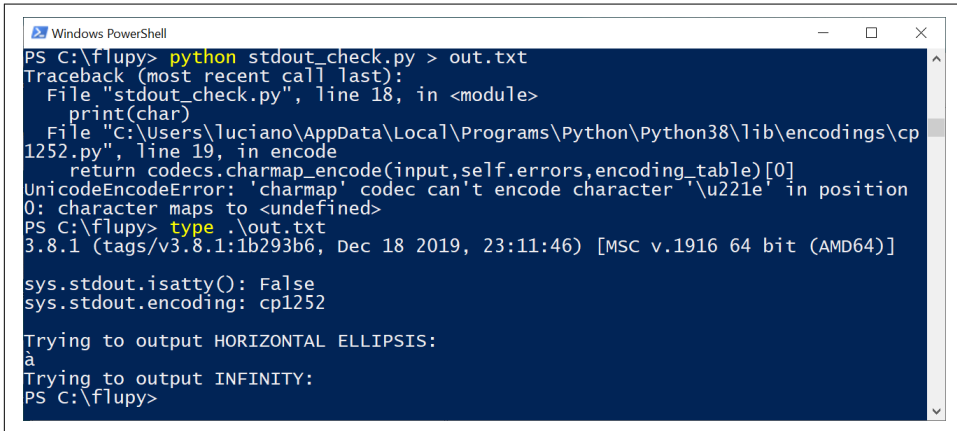
sys.stdout.isatty(): True
sys.stdout.encoding: utf-8

Trying to output HORIZONTAL ELLIPSIS:
...
Trying to output INFINITY:
∞
Trying to output CIRCLED NUMBER FORTY TWO:
Ⓣ
PS C:\flupy>
```

Figure 4-3. Running stdout_check.py on PowerShell.

Despite `chcp` reporting the active code as 437, `sys.stdout.encoding` is UTF-8, so the HORIZONTAL ELLIPSIS and INFINITY both output correctly. The CIRCLED NUMBER FORTY TWO is replaced by a rectangle, but no error is raised. Presumably it is recognized as a valid character, but the console font doesn't have the glyph to display it.

However, when I redirect the output of `stdout_check.py` to a file, I get [Figure 4-4](#).



```
PS C:\flupy> python stdout_check.py > out.txt
Traceback (most recent call last):
  File "stdout_check.py", line 18, in <module>
    print(char)
  File "C:\Users\luciano\AppData\Local\Programs\Python\Python38\lib\encodings\cp
1252.py", line 19, in encode
    return codecs.charmap_encode(input,self.errors,encoding_table)[0]
UnicodeEncodeError: 'charmap' codec can't encode character '\u221e' in position
0: character maps to <undefined>
PS C:\flupy> type .\out.txt
3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)]

sys.stdout.isatty(): False
sys.stdout.encoding: cp1252

Trying to output HORIZONTAL ELLIPSIS:
à
Trying to output INFINITY:
PS C:\flupy>
```

Figure 4-4. Running `stdout_check.py` on PowerShell, redirecting output.

The first problem demonstrated by [Figure 4-4](#) is the `UnicodeEncodeError` mentioning character `'\u221e'`, because `sys.stdout.encoding` is `'cp1252'`—a code page that doesn't have the INFINITY character.

Reading `out.txt` with the `type` command—or a Windows editor like VS Code or Sublime Text—shows that instead of HORIZONTAL ELLIPSIS, I got `'à'` (LATIN SMALL LETTER A WITH GRAVE). As it turns out, the byte value `0x85` in CP 1252 means `'...'`, but in CP 437 the same byte value represents `'à'`. So it seems the active code page does matter, not in a sensible or useful way, but as partial explanation of a bad Unicode experience.



I used a laptop configured for the US market, running Windows 10 OEM to run these experiments. Windows versions localized for other countries may have different encoding configurations. For example, in Brazil the Windows console uses code page 850 by default—not 437.

To wrap up this maddening issue of default encodings, let's give a final look at the different encodings in [Example 4-11](#):

- If you omit the encoding argument when opening a file, the default is given by `locale.getpreferredencoding()` ('cp1252' in [Example 4-11](#)).
- The encoding of `sys.stdout|stdin|stderr` used to be set by the `PYTHONIOENCODING` environment variable before Python 3.6—now that variable is ignored, unless `PYTHONLEGACYWINDOWSSDIO` is set to a nonempty string. Otherwise, the encoding for standard I/O is UTF-8 for interactive I/O, or defined by `locale.getpreferredencoding()` if the output/input is redirected to/from a file.
- `sys.getdefaultencoding()` is used internally by Python in implicit conversions of binary data to/from `str`. Changing this setting is not supported.
- `sys.getfilesystemencoding()` is used to encode/decode filenames (not file contents). It is used when `open()` gets a `str` argument for the filename; if the filename is given as a `bytes` argument, it is passed unchanged to the OS API.



On GNU/Linux and macOS, all of these encodings are set to UTF-8 by default, and have been for several years, so I/O handles all Unicode characters. On Windows, not only are different encodings used in the same system, but they are usually code pages like 'cp850' or 'cp1252' that support only ASCII, with 127 additional characters that are not the same from one encoding to the other. Therefore, Windows users are far more likely to face encoding errors unless they are extra careful.

To summarize, the most important encoding setting is that returned by `locale.getpreferredencoding()`: it is the default for opening text files and for `sys.stdout/stdin/stderr` when they are redirected to files. However, the [documentation](#) reads (in part):

```
locale.getpreferredencoding(do_setlocale=True)
```

Return the encoding used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess. [...]

Therefore, the best advice about encoding defaults is: do not rely on them.

You will avoid a lot of pain if you follow the advice of the Unicode sandwich and always are explicit about the encodings in your programs. Unfortunately, Unicode is painful even if you get your `bytes` correctly converted to `str`. The next two sections cover subjects that are simple in ASCII-land, but get quite complex on planet Unicode: text normalization (i.e., converting text to a uniform representation for comparisons) and sorting.

Normalizing Unicode for Reliable Comparisons

String comparisons are complicated by the fact that Unicode has combining characters: diacritics and other marks that attach to the preceding character, appearing as one when printed.

For example, the word “café” may be composed in two ways, using four or five code points, but the result looks exactly the same:

```
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1, s2
('café', 'café')
>>> len(s1), len(s2)
(4, 5)
>>> s1 == s2
False
```

Placing COMBINING ACUTE ACCENT (U+0301) after “e” renders “é”. In the Unicode standard, sequences like 'é' and 'e\u0301' are called “canonical equivalents,” and applications are supposed to treat them as the same. But Python sees two different sequences of code points, and considers them not equal.

The solution is `unicodedata.normalize()`. The first argument to that function is one of four strings: 'NFC', 'NFD', 'NFKC', and 'NFKD'. Let's start with the first two.

Normalization Form C (NFC) composes the code points to produce the shortest equivalent string, while NFD decomposes, expanding composed characters into base characters and separate combining characters. Both of these normalizations make comparisons work as expected, as the next example shows:

```
>>> from unicodedata import normalize
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> len(s1), len(s2)
(4, 5)
>>> len(normalize('NFC', s1)), len(normalize('NFC', s2))
(4, 4)
>>> len(normalize('NFD', s1)), len(normalize('NFD', s2))
(5, 5)
>>> normalize('NFC', s1) == normalize('NFC', s2)
True
>>> normalize('NFD', s1) == normalize('NFD', s2)
True
```

Keyboard drivers usually generate composed characters, so text typed by users will be in NFC by default. However, to be safe, it may be good to normalize strings with `normalize('NFC', user_text)` before saving. NFC is also the normalization form recommended by the W3C in “[Character Model for the World Wide Web: String Matching and Searching](#)”.

Some single characters are normalized by NFC into another single character. The symbol for the ohm (Ω) unit of electrical resistance is normalized to the Greek upper-case omega. They are visually identical, but they compare as unequal, so it is essential to normalize to avoid surprises:

```
>>> from unicodedata import normalize, name
>>> ohm = '\u2126'
>>> name(ohm)
'OHM SIGN'
>>> ohm_c = normalize('NFC', ohm)
>>> name(ohm_c)
'GREEK CAPITAL LETTER OMEGA'
>>> ohm == ohm_c
False
>>> normalize('NFC', ohm) == normalize('NFC', ohm_c)
True
```

The other two normalization forms are NFKC and NFKD, where the letter K stands for “compatibility.” These are stronger forms of normalization, affecting the so-called “compatibility characters.” Although one goal of Unicode is to have a single “canonical” code point for each character, some characters appear more than once for compatibility with preexisting standards. For example, the MICRO SIGN, μ (U+00B5), was added to Unicode to support round-trip conversion to latin1, which includes it, even though the same character is part of the Greek alphabet with code point U+03BC (GREEK SMALL LETTER MU). So, the micro sign is considered a “compatibility character.”

In the NFKC and NFKD forms, each compatibility character is replaced by a “compatibility decomposition” of one or more characters that are considered a “preferred” representation, even if there is some formatting loss—ideally, the formatting should be the responsibility of external markup, not part of Unicode. To exemplify, the compatibility decomposition of the one-half fraction '½' (U+00BD) is the sequence of three characters '1/2', and the compatibility decomposition of the micro sign 'μ' (U+00B5) is the lowercase mu 'μ' (U+03BC).⁷

Here is how the NFKC works in practice:

```
>>> from unicodedata import normalize, name
>>> half = '\N{VULGAR FRACTION ONE HALF}'
>>> print(half)
½
>>> normalize('NFKC', half)
'1/2'
```

⁷ Curiously, the micro sign is considered a “compatibility character,” but the ohm symbol is not. The end result is that NFC doesn’t touch the micro sign but changes the ohm symbol to capital omega, while NFKC and NFKD change both the ohm and the micro into Greek characters.

```

>>> for char in normalize('NFKC', half):
...     print(char, name(char), sep='\t')
...
1 DIGIT ONE
/ FRACTION SLASH
2 DIGIT TWO
>>> four_squared = '4²'
>>> normalize('NFKC', four_squared)
'42'
>>> micro = 'μ'
>>> micro_kc = normalize('NFKC', micro)
>>> micro, micro_kc
('μ', 'μ')
>>> ord(micro), ord(micro_kc)
(181, 956)
>>> name(micro), name(micro_kc)
('MICRO SIGN', 'GREEK SMALL LETTER MU')

```

Although '1/2' is a reasonable substitute for '½', and the micro sign is really a lowercase Greek mu, converting '4²' to '42' changes the meaning. An application could store '4²' as '4²', but the `normalize` function knows nothing about formatting. Therefore, NFKC or NFKD may lose or distort information, but they can produce convenient intermediate representations for searching and indexing.

Unfortunately, with Unicode everything is always more complicated than it first seems. For the VULGAR FRACTION ONE HALF, the NFKC normalization produced 1 and 2 joined by FRACTION SLASH, instead of SOLIDUS, a.k.a. “slash”—the familiar character with ASCII code decimal 47. Therefore, searching for the three-character ASCII sequence '1/2' would not find the normalized Unicode sequence.



NFKC and NFKD normalization cause data loss and should be applied only in special cases like search and indexing, and not for permanent storage of text.

When preparing text for searching or indexing, another operation is useful: case folding, our next subject.

Case Folding

Case folding is essentially converting all text to lowercase, with some additional transformations. It is supported by the `str.casefold()` method.

For any string `s` containing only latin1 characters, `s.casefold()` produces the same result as `s.lower()`, with only two exceptions—the micro sign 'μ' is changed to the

Greek lowercase mu (which looks the same in most fonts) and the German Eszett or “sharp s” (ß) becomes “ss”:

```
>>> micro = 'μ'
>>> name(micro)
'MICRO SIGN'
>>> micro_cf = micro.casefold()
>>> name(micro_cf)
'GREEK SMALL LETTER MU'
>>> micro, micro_cf
('μ', 'μ')
>>> eszett = 'ß'
>>> name(eszett)
'LATIN SMALL LETTER SHARP S'
>>> eszett_cf = eszett.casefold()
>>> eszett, eszett_cf
('ß', 'ss')
```

There are nearly 300 code points for which `str.casefold()` and `str.lower()` return different results.

As usual with anything related to Unicode, case folding is a hard issue with plenty of linguistic special cases, but the Python core team made an effort to provide a solution that hopefully works for most users.

In the next couple of sections, we’ll put our normalization knowledge to use developing utility functions.

Utility Functions for Normalized Text Matching

As we’ve seen, NFC and NFD are safe to use and allow sensible comparisons between Unicode strings. NFC is the best normalized form for most applications. `str.casefold()` is the way to go for case-insensitive comparisons.

If you work with text in many languages, a pair of functions like `nfc_equal` and `fold_equal` in [Example 4-13](#) are useful additions to your toolbox.

Example 4-13. `normeq.py`: normalized Unicode string comparison

```
"""
Utility functions for normalized Unicode string comparison.

Using Normal Form C, case sensitive:

>>> s1 = 'café'
>>> s2 = 'cafe\u00301'
>>> s1 == s2
False
>>> nfc_equal(s1, s2)
True
```

```
>>> nfc_equal('A', 'a')
False
```

Using Normal Form C with case folding:

```
>>> s3 = 'Straße'
>>> s4 = 'strasse'
>>> s3 == s4
False
>>> nfc_equal(s3, s4)
False
>>> fold_equal(s3, s4)
True
>>> fold_equal(s1, s2)
True
>>> fold_equal('A', 'a')
True
```

```
"""
```

```
from unicodedata import normalize

def nfc_equal(str1, str2):
    return normalize('NFC', str1) == normalize('NFC', str2)

def fold_equal(str1, str2):
    return (normalize('NFC', str1).casefold() ==
            normalize('NFC', str2).casefold())
```

Beyond Unicode normalization and case folding—which are both part of the Unicode standard—sometimes it makes sense to apply deeper transformations, like changing 'café' into 'cafe'. We'll see when and how in the next section.

Extreme “Normalization”: Taking Out Diacritics

The Google Search secret sauce involves many tricks, but one of them apparently is ignoring diacritics (e.g., accents, cedillas, etc.), at least in some contexts. Removing diacritics is not a proper form of normalization because it often changes the meaning of words and may produce false positives when searching. But it helps coping with some facts of life: people sometimes are lazy or ignorant about the correct use of diacritics, and spelling rules change over time, meaning that accents come and go in living languages.

Outside of searching, getting rid of diacritics also makes for more readable URLs, at least in Latin-based languages. Take a look at the URL for the Wikipedia article about the city of São Paulo:

```
https://en.wikipedia.org/wiki/S%C3%A3o\_Paulo
```

The %C3%A3 part is the URL-escaped, UTF-8 rendering of the single letter “ã” (“a” with tilde). The following is much easier to recognize, even if it is not the right spelling:

`https://en.wikipedia.org/wiki/Sao_Paulo`

To remove all diacritics from a str, you can use a function like [Example 4-14](#).

Example 4-14. simplify.py: function to remove all combining marks

```
import unicodedata
import string

def shave_marks(txt):
    """Remove all diacritic marks"""
    norm_txt = unicodedata.normalize('NFD', txt) ❶
    shaved = ''.join(c for c in norm_txt
                     if not unicodedata.combining(c)) ❷
    return unicodedata.normalize('NFC', shaved) ❸
```

- ❶ Decompose all characters into base characters and combining marks.
- ❷ Filter out all combining marks.
- ❸ Recompose all characters.

[Example 4-15](#) shows a couple of uses of `shave_marks`.

Example 4-15. Two examples using shave_marks from [Example 4-14](#)

```
>>> order = "Herr Voß: • ½ cup of Ætker™ caffè latte • bowl of açaí."
>>> shave_marks(order)
'Herr Voß: • ½ cup of Ætke™ caffè latte • bowl of acai.' ❶
>>> Greek = 'Ζέφυρος, Ζέφиро'
>>> shave_marks(Greek)
'Ζεφυρος, Zefiro' ❷
```

- ❶ Only the letters “è”, “ç”, and “í” were replaced.
- ❷ Both “ξ” and “é” were replaced.

The function `shave_marks` from [Example 4-14](#) works all right, but maybe it goes too far. Often the reason to remove diacritics is to change Latin text to pure ASCII, but `shave_marks` also changes non-Latin characters—like Greek letters—which will never become ASCII just by losing their accents. So it makes sense to analyze each base

character and to remove attached marks only if the base character is a letter from the Latin alphabet. This is what [Example 4-16](#) does.

Example 4-16. Function to remove combining marks from Latin characters (import statements are omitted as this is part of the `simplify.py` module from [Example 4-14](#))

```
def shave_marks_latin(txt):  
    """Remove all diacritic marks from Latin base characters"""  
    norm_txt = unicodedata.normalize('NFD', txt) ❶  
    latin_base = False  
    preserve = []  
    for c in norm_txt:  
        if unicodedata.combining(c) and latin_base: ❷  
            continue # ignore diacritic on Latin base char  
        preserve.append(c) ❸  
        # if it isn't a combining char, it's a new base char  
        if not unicodedata.combining(c): ❹  
            latin_base = c in string.ascii_letters  
    shaved = ''.join(preserve)  
    return unicodedata.normalize('NFC', shaved) ❺
```

- 1 Decompose all characters into base characters and combining marks.
- 2 Skip over combining marks when base character is Latin.
- 3 Otherwise, keep current character.
- 4 Detect new base character and determine if it's Latin.
- 5 Recompose all characters.

An even more radical step would be to replace common symbols in Western texts (e.g., curly quotes, em dashes, bullets, etc.) into ASCII equivalents. This is what the function `asciize` does in [Example 4-17](#).

Example 4-17. Transform some Western typographical symbols into ASCII (this snippet is also part of `simplify.py` from [Example 4-14](#))

[illegible]

```
multi_map = str.maketrans({
    '€': 'EUR',
    '...': '...',
    'Æ': 'AE',
    'æ': 'ae',
    'Œ': 'OE',
    'œ': 'oe',
```



```

    '™': '(TM)',
    '‰': '<per mille>',
    '†': '**',
    '‡': '***',
})

multi_map.update(single_map) ❸

def dewinize(txt):
    """Replace Win1252 symbols with ASCII chars or sequences"""
    return txt.translate(multi_map) ❹

def asciize(txt):
    no_marks = shave_marks_latin(dewinize(txt)) ❺
    no_marks = no_marks.replace('ß', 'ss') ❻
    return unicodedata.normalize('NFKC', no_marks) ❼

```

- ❶ Build mapping table for char-to-char replacement.
- ❷ Build mapping table for char-to-string replacement.
- ❸ Merge mapping tables.
- ❹ dewinize does not affect ASCII or latin1 text, only the Microsoft additions to latin1 in cp1252.
- ❺ Apply dewinize and remove diacritical marks.
- ❻ Replace the Eszett with “ss” (we are not using case fold here because we want to preserve the case).
- ❼ Apply NFKC normalization to compose characters with their compatibility code points.

Example 4-18 shows asciize in use.

Example 4-18. Two examples using asciize from Example 4-17

```

>>> order = "Herr Voß: • ½ cup of ☿tker™ caffè latte • bowl of açai."
>>> dewinize(order)
'Herr Voß: - ½ cup of 0Etker(TM) caffè latte - bowl of açai.' ❶
>>> asciize(order)
'Herr Voss: - 1/2 cup of 0Etker(TM) caffè latte - bowl of acai.' ❷

```

- ❶ `dewinize` replaces curly quotes, bullets, and [™] (trademark symbol).
- ❷ `asciize` applies `dewinize`, drops diacritics, and replaces the 'ß'.



Different languages have their own rules for removing diacritics. For example, Germans change the 'ü' into 'ue'. Our `asciize` function is not as refined, so it may or not be suitable for your language. It works acceptably for Portuguese, though.

To summarize, the functions in *simplify.py* go way beyond standard normalization and perform deep surgery on the text, with a good chance of changing its meaning. Only you can decide whether to go so far, knowing the target language, your users, and how the transformed text will be used.

This wraps up our discussion of normalizing Unicode text.

Now let's sort out Unicode sorting.

Sorting Unicode Text

Python sorts sequences of any type by comparing the items in each sequence one by one. For strings, this means comparing the code points. Unfortunately, this produces unacceptable results for anyone who uses non-ASCII characters.

Consider sorting a list of fruits grown in Brazil:

```
>>> fruits = ['caju', 'atemoia', 'cajá', 'açai', 'acerola']
>>> sorted(fruits)
['acerola', 'atemoia', 'açai', 'caju', 'cajá']
```

Sorting rules vary for different locales, but in Portuguese and many languages that use the Latin alphabet, accents and cedillas rarely make a difference when sorting.⁸ So “cajá” is sorted as “caja,” and must come before “caju.”

The sorted fruits list should be:

```
['açai', 'acerola', 'atemoia', 'cajá', 'caju']
```

The standard way to sort non-ASCII text in Python is to use the `locale.strxfrm` function which, according to the [locale module docs](#), “transforms a string to one that can be used in locale-aware comparisons.”

⁸ Diacritics affect sorting only in the rare case when they are the only difference between two words—in that case, the word with a diacritic is sorted after the plain word.

To enable `locale.strxfrm`, you must first set a suitable locale for your application, and pray that the OS supports it. The sequence of commands in [Example 4-19](#) may work for you.

Example 4-19. `locale_sort.py`: using the `locale.strxfrm` function as the sort key

```
import locale
my_locale = locale.setlocale(locale.LC_COLLATE, 'pt_BR.UTF-8')
print(my_locale)
fruits = ['caju', 'atemoia', 'cajá', 'açai', 'acerola']
sorted_fruits = sorted(fruits, key=locale.strxfrm)
print(sorted_fruits)
```

Running [Example 4-19](#) on GNU/Linux (Ubuntu 19.10) with the `pt_BR.UTF-8` locale installed, I get the correct result:

```
'pt_BR.UTF-8'
['açai', 'acerola', 'atemoia', 'cajá', 'caju']
```

So you need to call `setlocale(LC_COLLATE, «your_locale»)` before using `locale.strxfrm` as the key when sorting.

There are some caveats, though:

- Because locale settings are global, calling `setlocale` in a library is not recommended. Your application or framework should set the locale when the process starts, and should not change it afterward.
- The locale must be installed on the OS, otherwise `setlocale` raises a `locale.Error: unsupported locale setting` exception.
- You must know how to spell the locale name.
- The locale must be correctly implemented by the makers of the OS. I was successful on Ubuntu 19.10, but not on macOS 10.14. On macOS, the call `setlocale(LC_COLLATE, 'pt_BR.UTF-8')` returns the string `'pt_BR.UTF-8'` with no complaints. But `sorted(fruits, key=locale.strxfrm)` produced the same incorrect result as `sorted(fruits)` did. I also tried the `fr_FR`, `es_ES`, and `de_DE` locales on macOS, but `locale.strxfrm` never did its job.⁹

So the standard library solution to internationalized sorting works, but seems to be well supported only on GNU/Linux (perhaps also on Windows, if you are an expert). Even then, it depends on locale settings, creating deployment headaches.

⁹ Again, I could not find a solution, but did find other people reporting the same problem. Alex Martelli, one of the tech reviewers, had no problem using `setlocale` and `locale.strxfrm` on his Macintosh with macOS 10.9. In summary: your mileage may vary.

Fortunately, there is a simpler solution: the *pyuca* library, available on *PyPI*.

Sorting with the Unicode Collation Algorithm

James Tauber, prolific Django contributor, must have felt the pain and created *pyuca*, a pure-Python implementation of the Unicode Collation Algorithm (UCA). **Example 4-20** shows how easy it is to use.

Example 4-20. Using the `pyuca.Collator.sort_key` method

```
>>> import pyuca
>>> coll = pyuca.Collator()
>>> fruits = ['caju', 'atemoia', 'cajá', 'açai', 'acerola']
>>> sorted_fruits = sorted(fruits, key=coll.sort_key)
>>> sorted_fruits
['açai', 'acerola', 'atemoia', 'cajá', 'caju']
```

This is simple and works on GNU/Linux, macOS, and Windows, at least with my small sample.

pyuca does not take the locale into account. If you need to customize the sorting, you can provide the path to a custom collation table to the `Collator()` constructor. Out of the box, it uses *allkeys.txt*, which is bundled with the project. That's just a copy of the **Default Unicode Collation Element Table from *Unicode.org***.



PyICU: Miro's Recommendation for Unicode Sorting

(Tech reviewer Miroslav Šedivý is a polyglot and an expert on Unicode. This is what he wrote about *pyuca*.)

pyuca has one sorting algorithm that does not respect the sorting order in individual languages. For instance, Ä in German is between A and B, while in Swedish it comes after Z. Have a look at **PyICU** that works like locale without changing the locale of the process. It is also needed if you want to change the case of iİ/iI in Turkish. PyICU includes an extension that must be compiled, so it may be harder to install in some systems than *pyuca*, which is just Python.

By the way, that collation table is one of the many data files that comprise the Unicode database, our next subject.

The Unicode Database

The Unicode standard provides an entire database—in the form of several structured text files—that includes not only the table mapping code points to character names,

but also metadata about the individual characters and how they are related. For example, the Unicode database records whether a character is printable, is a letter, is a decimal digit, or is some other numeric symbol. That’s how the `str` methods `isalpha`, `isprintable`, `isdecimal`, and `isnumeric` work. `str.casefold` also uses information from a Unicode table.



The `unicodedata.category(char)` function returns the two-letter category of `char` from the Unicode database. The higher-level `str` methods are easier to use. For example, `label.isalpha()` returns `True` if every character in `label` belongs to one of these categories: `Lm`, `Lt`, `Lu`, `Ll`, or `Lo`. To learn what those codes mean, see “[General Category](#)” in the English Wikipedia’s “[Unicode character property](#)” article.

Finding Characters by Name

The `unicodedata` module has functions to retrieve character metadata, including `unicodedata.name()`, which returns a character’s official name in the standard. [Figure 4-5](#) demonstrates that function.¹⁰

```
>>> from unicodedata import name
>>> name('A')
'LATIN CAPITAL LETTER A'
>>> name('ã')
'LATIN SMALL LETTER A WITH TILDE'
>>> name('♚')
'BLACK CHESS QUEEN'
>>> name('😺')
'GRINNING CAT FACE WITH SMILING EYES'
```

Figure 4-5. Exploring `unicodedata.name()` in the Python console.

You can use the `name()` function to build apps that let users search for characters by name. [Figure 4-6](#) demonstrates the `cf.py` command-line script that takes one or more words as arguments, and lists the characters that have those words in their official Unicode names. The full source code for `cf.py` is in [Example 4-21](#).

¹⁰ That’s an image—not a code listing—because emojis are not well supported by O’Reilly’s digital publishing toolchain as I write this.

```

$ ./cf.py cat smiling
U+1F638 🐱 GRINNING CAT FACE WITH SMILING EYES
U+1F63A 🐱 SMILING CAT FACE WITH OPEN MOUTH
U+1F63B 🐱 SMILING CAT FACE WITH HEART-SHAPED EYES

```

Figure 4-6. Using *cf.py* to find smiling cats.



Emoji support varies widely across operating systems and apps. In recent years the macOS terminal offers the best support for emojis, followed by modern GNU/Linux graphic terminals. Windows *cmd.exe* and PowerShell now support Unicode output, but as I write this section in January 2020, they still don't display emojis—at least not “out of the box.” Tech reviewer Leonardo Rochaël told me about a new, open source [Windows Terminal by Microsoft](#), which may have better Unicode support than the older Microsoft consoles. I did not have time to try it.

In [Example 4-21](#), note the `if` statement in the `find` function using the `.issubset()` method to quickly test whether all the words in the query set appear in the list of words built from the character's name. Thanks to Python's rich set API, we don't need a nested `for` loop and another `if` to implement this check.

Example 4-21. *cf.py*: the character finder utility

```

#!/usr/bin/env python3
import sys
import unicodedata

START, END = ord(' '), sys.maxunicode + 1 ❶

def find(*query_words, start=START, end=END): ❷
    query = {w.upper() for w in query_words} ❸
    for code in range(start, end):
        char = chr(code) ❹
        name = unicodedata.name(char, None) ❺
        if name and query.issubset(name.split()): ❻
            print(f'U+{code:04X}\t{char}\t{name}') ❼

def main(words):
    if words:
        find(*words)
    else:
        print('Please provide words to find.')

if __name__ == '__main__':
    main(sys.argv[1:])

```

- ❶ Set defaults for the range of code points to search.
- ❷ `find` accepts `query_words` and optional keyword-only arguments to limit the range of the search, to facilitate testing.
- ❸ Convert `query_words` into a set of uppercased strings.
- ❹ Get the Unicode character for code.
- ❺ Get the name of the character, or `None` if the code point is unassigned.
- ❻ If there is a name, split it into a list of words, then check that the query set is a subset of that list.
- ❼ Print out line with code point in U+9999 format, the character, and its name.

The `unicodedata` module has other interesting functions. Next, we'll see a few that are related to getting information from characters that have numeric meaning.

Numeric Meaning of Characters

The `unicodedata` module includes functions to check whether a Unicode character represents a number and, if so, its numeric value for humans—as opposed to its code point number. [Example 4-22](#) shows the use of `unicodedata.name()` and `unicodedata.numeric()`, along with the `.isdecimal()` and `.isnumeric()` methods of `str`.

Example 4-22. Demo of Unicode database numerical character metadata (callouts describe each column in the output)

```
import unicodedata
import re

re_digit = re.compile(r'\d')

sample = '1\xbc\xb2\u0969\u136b\u216b\u2466\u2480\u3285'

for char in sample:
    print(f'U+{ord(char):04x}',          ❶
          char.center(6),                ❷
          're_dig' if re_digit.match(char) else '-', ❸
          'isdig' if char.isdigit() else '-',        ❹
          'isnum' if char.isnumeric() else '-',      ❺
          f'{unicodedata.numeric(char):5.2f}',       ❻
          unicodedata.name(char),                 ❼
          sep='\t')
```

- ❶ Code point in U+0000 format.
- ❷ Character centralized in a str of length 6.
- ❸ Show `re_dig` if character matches the `r'\d'` regex.
- ❹ Show `isdig` if `char.isdigit()` is True.
- ❺ Show `isnum` if `char.isnumeric()` is True.
- ❻ Numeric value formatted with width 5 and 2 decimal places.
- ❼ Unicode character name.

Running **Example 4-22** gives you **Figure 4-7**, if your terminal font has all those glyphs.

```

$ python3 numerics_demo.py
U+0031  1      re_dig isdig isnum  1.00  DIGIT ONE
U+00bc  ¼      -      isdig isnum  0.25  VULGAR FRACTION ONE QUARTER
U+00b2  ²      -      isdig isnum  2.00  SUPERScript TWO
U+0969  ३      re_dig isdig isnum  3.00  DEVANAGARI DIGIT THREE
U+136b  ፫      -      isdig isnum  3.00  ETHIOPIc DIGIT THREE
U+216b  XII     -      isdig isnum 12.00  ROMAN NUMERAL TWELVE
U+2466  ⑦      -      isdig isnum  7.00  CIRCLED DIGIT SEVEN
U+2480  ⑬      -      isdig isnum 13.00  PARENTHESIZED NUMBER THIRTEEN
U+3285  ⑆      -      isdig isnum  6.00  CIRCLED IDEOGRAPH SIX
$

```

Figure 4-7. macOS terminal showing numeric characters and metadata about them; `re_dig` means the character matches the regular expression `r'\d'`.

The sixth column of **Figure 4-7** is the result of calling `unicodedata.numeric(char)` on the character. It shows that Unicode knows the numeric value of symbols that represent numbers. So if you want to create a spreadsheet application that supports Tamil digits or Roman numerals, go for it!

Figure 4-7 shows that the regular expression `r'\d'` matches the digit “1” and the Devanagari digit 3, but not some other characters that are considered digits by the `isdigit` function. The `re` module is not as savvy about Unicode as it could be. The new regex module available on PyPI was designed to eventually replace `re` and provides better Unicode support.¹¹ We’ll come back to the `re` module in the next section.

¹¹ Although it was not better than `re` at identifying digits in this particular sample.

Throughout this chapter we've used several `unicodedata` functions, but there are many more we did not cover. See the standard library documentation for the [unicode data module](#).

Next we'll take a quick look at dual-mode APIs offering functions that accept `str` or `bytes` arguments with special handling depending on the type.

Dual-Mode `str` and `bytes` APIs

Python's standard library has functions that accept `str` or `bytes` arguments and behave differently depending on the type. Some examples can be found in the `re` and `os` modules.

`str` Versus `bytes` in Regular Expressions

If you build a regular expression with `bytes`, patterns such as `\d` and `\w` only match ASCII characters; in contrast, if these patterns are given as `str`, they match Unicode digits or letters beyond ASCII. [Example 4-23](#) and [Figure 4-8](#) compare how letters, ASCII digits, superscripts, and Tamil digits are matched by `str` and `bytes` patterns.

Example 4-23. `ramanujan.py`: compare behavior of simple `str` and `bytes` regular expressions

```
import re

re_numbers_str = re.compile(r'\d+') ❶
re_words_str = re.compile(r'\w+')
re_numbers_bytes = re.compile(rb'\d+') ❷
re_words_bytes = re.compile(rb'\w+')

text_str = ("Ramanujan saw \u0be7\u0bed\u0be8\u0bef" ❸
            " as 1729 = 13 + 123 = 93 + 103." ) ❹

text_bytes = text_str.encode('utf_8') ❺

print(f'Text\n {text_str!r}')
print('Numbers')
print(' str :', re_numbers_str.findall(text_str)) ❻
print(' bytes:', re_numbers_bytes.findall(text_bytes)) ❼
print('Words')
print(' str :', re_words_str.findall(text_str)) ❽
print(' bytes:', re_words_bytes.findall(text_bytes)) ❾
```

❶ The first two regular expressions are of the `str` type.

❷ The last two are of the `bytes` type.

- ③ Unicode text to search, containing the Tamil digits for 1729 (the logical line continues until the right parenthesis token).
- ④ This string is joined to the previous one at compile time (see “2.4.2. String literal concatenation” in *The Python Language Reference*).
- ⑤ A bytes string is needed to search with the bytes regular expressions.
- ⑥ The `str` pattern `r'\d+'` matches the Tamil and ASCII digits.
- ⑦ The bytes pattern `rb'\d+'` matches only the ASCII bytes for digits.
- ⑧ The `str` pattern `r'\w+'` matches the letters, superscripts, Tamil, and ASCII digits.
- ⑨ The bytes pattern `rb'\w+'` matches only the ASCII bytes for letters and digits.

```

1. bash
$ python3 ramanujan.py
Text
'Ramanujan saw க௭௨௯ as 1729 = 1³ + 12³ = 9³ + 10³.'
Numbers
str : ['க௭௨௯', '1729', '1', '12', '9', '10']
bytes: [b'1729', b'1', b'12', b'9', b'10']
Words
str : ['Ramanujan', 'saw', 'க௭௨௯', 'as', '1729', '1³', '12³', '9³', '10³']
bytes: [b'Ramanujan', b'saw', b'as', b'1729', b'1', b'12', b'9', b'10']
$

```

Figure 4-8. Screenshot of running `ramanujan.py` from *Example 4-23*.

Example 4-23 is a trivial example to make one point: you can use regular expressions on `str` and `bytes`, but in the second case, bytes outside the ASCII range are treated as nondigits and nonword characters.

For `str` regular expressions, there is a `re.ASCII` flag that makes `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s`, and `\S` perform ASCII-only matching. See the [documentation of the `re` module](#) for full details.

Another important dual-mode module is `os`.

str Versus bytes in os Functions

The GNU/Linux kernel is not Unicode savvy, so in the real world you may find filenames made of byte sequences that are not valid in any sensible encoding scheme, and cannot be decoded to `str`. File servers with clients using a variety of OSes are particularly prone to this problem.

In order to work around this issue, all `os` module functions that accept filenames or pathnames take arguments as `str` or `bytes`. If one such function is called with a `str` argument, the argument will be automatically converted using the codec named by `sys.getfilesystemencoding()`, and the OS response will be decoded with the same codec. This is almost always what you want, in keeping with the Unicode sandwich best practice.

But if you must deal with (and perhaps fix) filenames that cannot be handled in that way, you can pass `bytes` arguments to the `os` functions to get `bytes` return values. This feature lets you deal with any file or pathname, no matter how many gremlins you may find. See [Example 4-24](#).

Example 4-24. `listdir` with `str` and `bytes` arguments and results

```
>>> os.listdir('.') ❶
['abc.txt', 'digits-of-π.txt']
>>> os.listdir(b'.') ❷
[b'abc.txt', b'digits-of-\xcf\x80.txt']
```

- ❶ The second filename is “digits-of- π .txt” (with the Greek letter pi).
- ❷ Given a byte argument, `listdir` returns filenames as bytes: `b'\xcf\x80'` is the UTF-8 encoding of the Greek letter pi.

To help with manual handling of `str` or `bytes` sequences that are filenames or pathnames, the `os` module provides special encoding and decoding functions `os.fsencode(name_or_path)` and `os.fsdecode(name_or_path)`. Both of these functions accept an argument of type `str`, `bytes`, or an object implementing the `os.PathLike` interface since Python 3.6.

Unicode is a deep rabbit hole. Time to wrap up our exploration of `str` and `bytes`.

Chapter Summary

We started the chapter by dismissing the notion that `1 character == 1 byte`. As the world adopts Unicode, we need to keep the concept of text strings separated from the binary sequences that represent them in files, and Python 3 enforces this separation.

After a brief overview of the binary sequence data types—`bytes`, `bytearray`, and `memoryview`—we jumped into encoding and decoding, with a sampling of important codecs, followed by approaches to prevent or deal with the infamous `UnicodeEncodeError`, `UnicodeDecodeError`, and the `SyntaxError` caused by wrong encoding in Python source files.

We then considered the theory and practice of encoding detection in the absence of metadata: in theory, it can't be done, but in practice the Chardet package pulls it off pretty well for a number of popular encodings. Byte order marks were then presented as the only encoding hint commonly found in UTF-16 and UTF-32 files—sometimes in UTF-8 files as well.

In the next section, we demonstrated opening text files, an easy task except for one pitfall: the `encoding=` keyword argument is not mandatory when you open a text file, but it should be. If you fail to specify the encoding, you end up with a program that manages to generate “plain text” that is incompatible across platforms, due to conflicting default encodings. We then exposed the different encoding settings that Python uses as defaults and how to detect them. A sad realization for Windows users is that these settings often have distinct values within the same machine, and the values are mutually incompatible; GNU/Linux and macOS users, in contrast, live in a happier place where UTF-8 is the default pretty much everywhere.

Unicode provides multiple ways of representing some characters, so normalizing is a prerequisite for text matching. In addition to explaining normalization and case folding, we presented some utility functions that you may adapt to your needs, including drastic transformations like removing all accents. We then saw how to sort Unicode text correctly by leveraging the standard `locale` module—with some caveats—and an alternative that does not depend on tricky locale configurations: the external *pyuca* package.

We leveraged the Unicode database to program a command-line utility to search for characters by name—in 28 lines of code, thanks to the power of Python. We glanced at other Unicode metadata, and had a brief overview of dual-mode APIs where some functions can be called with `str` or `bytes` arguments, producing different results.

Further Reading

Ned Batchelder's 2012 PyCon US talk “[Pragmatic Unicode, or, How Do I Stop the Pain?](#)” was outstanding. Ned is so professional that he provides a full transcript of the talk along with the slides and video.

“Character encoding and Unicode in Python: How to (ノ ◕□◕)ノ へ ￣￣ with dignity” ([slides](#), [video](#)) was the excellent PyCon 2014 talk by Esther Nam and Travis Fischer, where I found this chapter's pithy epigraph: “Humans use text. Computers speak bytes.”

Lennart Regebro—one of the technical reviewers for the first edition of this book—shares his “Useful Mental Model of Unicode (UMMU)” in the short post “[Unconfusing Unicode: What Is Unicode?](#)”. Unicode is a complex standard, so Lennart's UMMU is a really useful starting point.

The official “[Unicode HOWTO](#)” in the Python docs approaches the subject from several different angles, from a good historic intro, to syntax details, codecs, regular expressions, filenames, and best practices for Unicode-aware I/O (i.e., the Unicode sandwich), with plenty of additional reference links from each section. [Chapter 4, “Strings](#)”, of Mark Pilgrim’s awesome book *[Dive into Python 3](#)* (Apress) also provides a very good intro to Unicode support in Python 3. In the same book, [Chapter 15](#) describes how the Chardet library was ported from Python 2 to Python 3, a valuable case study given that the switch from the old `str` to the new `bytes` is the cause of most migration pains, and that is a central concern in a library designed to detect encodings.

If you know Python 2 but are new to Python 3, Guido van Rossum’s “[What’s New in Python 3.0](#)” has 15 bullet points that summarize what changed, with lots of links. Guido starts with the blunt statement: “Everything you thought you knew about binary data and Unicode has changed.” Armin Ronacher’s blog post “[The Updated Guide to Unicode on Python](#)” is deep and highlights some of the pitfalls of Unicode in Python 3 (Armin is not a big fan of Python 3).

Chapter 2, “Strings and Text,” of the *[Python Cookbook, 3rd ed.](#)* (O’Reilly), by David Beazley and Brian K. Jones, has several recipes dealing with Unicode normalization, sanitizing text, and performing text-oriented operations on byte sequences. Chapter 5 covers files and I/O, and it includes “Recipe 5.17. Writing Bytes to a Text File,” showing that underlying any text file there is always a binary stream that may be accessed directly when needed. Later in the cookbook, the `struct` module is put to use in “Recipe 6.11. Reading and Writing Binary Arrays of Structures.”

Nick Coghlan’s “Python Notes” blog has two posts very relevant to this chapter: “[Python 3 and ASCII Compatible Binary Protocols](#)” and “[Processing Text Files in Python 3](#)”. Highly recommended.

A list of encodings supported by Python is available at “[Standard Encodings](#)” in the codecs module documentation. If you need to get that list programmatically, see how it’s done in the `/Tools/unicode/listcodecs.py` script that comes with the CPython source code.

The books *[Unicode Explained](#)* by Jukka K. Korpela (O’Reilly) and *[Unicode Demystified](#)* by Richard Gillam (Addison-Wesley) are not Python-specific but were very helpful as I studied Unicode concepts. *[Programming with Unicode](#)* by Victor Stinner is a free, self-published book (Creative Commons BY-SA) covering Unicode in general, as well as tools and APIs in the context of the main operating systems and a few programming languages, including Python.

The W3C pages “[Case Folding: An Introduction](#)” and “[Character Model for the World Wide Web: String Matching](#)” cover normalization concepts, with the former being a gentle introduction and the latter a working group note written in dry

standard-speak—the same tone of the “Unicode Standard Annex #15—Unicode Normalization Forms”. The “Frequently Asked Questions, Normalization” section from *Unicode.org* is more readable, as is the “NFC FAQ” by Mark Davis—author of several Unicode algorithms and president of the Unicode Consortium at the time of this writing.

In 2016, the Museum of Modern Art (MoMA) in New York added to its collection the original emoji, the 176 emojis designed by Shigetaka Kurita in 1999 for NTT DOCOMO—the Japanese mobile carrier. Going further back in history, *Emojipedia* published “Correcting the Record on the First Emoji Set”, crediting Japan’s SoftBank for the earliest known emoji set, deployed in cell phones in 1997. SoftBank’s set is the source of 90 emojis now in Unicode, including U+1F4A9 (PILE OF POO). Matthew Rothenberg’s *emojitracker.com* is a live dashboard showing counts of emoji usage on Twitter, updated in real time. As I write this, FACE WITH TEARS OF JOY (U+1F602) is the most popular emoji on Twitter, with more than 3,313,667,315 recorded occurrences.

Soapbox

Non-ASCII Names in Source Code: Should You Use Them?

Python 3 allows non-ASCII identifiers in source code:

```
>>> ação = 'PBR' # ação = stock
>>> ε = 10**-6 # ε = epsilon
```

Some people dislike the idea. The most common argument to stick with ASCII identifiers is to make it easy for everyone to read and edit code. That argument misses the point: you want your source code to be readable and editable by its intended audience, and that may not be “everyone.” If the code belongs to a multinational corporation or is open source and you want contributors from around the world, the identifiers should be in English, and then all you need is ASCII.

But if you are a teacher in Brazil, your students will find it easier to read code that uses Portuguese variable and function names, correctly spelled. And they will have no difficulty typing the cedillas and accented vowels on their localized keyboards.

Now that Python can parse Unicode names and UTF-8 is the default source encoding, I see no point in coding identifiers in Portuguese without accents, as we used to do in Python 2 out of necessity—unless you need the code to run on Python 2 also. If the names are in Portuguese, leaving out the accents won’t make the code more readable to anyone.

This is my point of view as a Portuguese-speaking Brazilian, but I believe it applies across borders and cultures: choose the human language that makes the code easier to read by the team, then use the characters needed for correct spelling.

What Is “Plain Text”?

For anyone who deals with non-English text on a daily basis, “plain text” does not imply “ASCII.” The [Unicode Glossary](#) defines *plain text* like this:

Computer-encoded text that consists only of a sequence of code points from a given standard, with no other formatting or structural information.

That definition starts very well, but I don’t agree with the part after the comma. HTML is a great example of a plain-text format that carries formatting and structural information. But it’s still plain text because every byte in such a file is there to represent a text character, usually using UTF-8. There are no bytes with nontext meaning, as you can find in a *.png* or *.xls* document where most bytes represent packed binary values like RGB values and floating-point numbers. In plain text, numbers are represented as sequences of digit characters.

I am writing this book in a plain-text format called—ironically—[AsciiDoc](#), which is part of the toolchain of O’Reilly’s excellent [Atlas book publishing platform](#). AsciiDoc source files are plain text, but they are UTF-8, not ASCII. Otherwise, writing this chapter would have been really painful. Despite the name, AsciiDoc is just great.

The world of Unicode is constantly expanding and, at the edges, tool support is not always there. Not all characters I wanted to show were available in the fonts used to render the book. That’s why I had to use images instead of listings in several examples in this chapter. On the other hand, the Ubuntu and macOS terminals display most Unicode text very well—including the Japanese characters for the word “mojibake”: 文字化け.

How Are str Code Points Represented in RAM?

The official Python docs avoid the issue of how the code points of a `str` are stored in memory. It is really an implementation detail. In theory, it doesn’t matter: whatever the internal representation, every `str` must be encoded to bytes on output.

In memory, Python 3 stores each `str` as a sequence of code points using a fixed number of bytes per code point, to allow efficient direct access to any character or slice.

Since Python 3.3, when creating a new `str` object, the interpreter checks the characters in it and chooses the most economic memory layout that is suitable for that particular `str`: if there are only characters in the `latin1` range, that `str` will use just one byte per code point. Otherwise, two or four bytes per code point may be used, depending on the `str`. This is a simplification; for the full details, look up [PEP 393—Flexible String Representation](#).

The flexible string representation is similar to the way the `int` type works in Python 3: if the integer fits in a machine word, it is stored in one machine word. Otherwise, the interpreter switches to a variable-length representation like that of the Python 2 `long` type. It is nice to see the spread of good ideas.

However, we can always count on Armin Ronacher to find problems in Python 3. He explained to me why that was not such a great idea in practice: it takes a single RAT (U+1F400) to inflate an otherwise all-ASCII text into a memory-hogging array using four bytes per character, when one byte would suffice for each character except the RAT. In addition, because of all the ways Unicode characters combine, the ability to quickly retrieve an arbitrary character by position is overrated—and extracting arbitrary slices from Unicode text is naïve at best, and often wrong, producing mojibake. As emojis become more popular, these problems will only get worse.