

1. I. Prologue
2. 1. The Python Data Model
 - a. What's new in this chapter
 - b. A Pythonic Card Deck
 - c. How Special Methods Are Used
 - i. Emulating Numeric Types
 - ii. String Representation
 - iii. Arithmetic Operators
 - iv. Boolean Value of a Custom Type
 - v. Collection API
 - d. Overview of Special Methods
 - e. Why len Is Not a Method
 - f. Chapter Summary
 - g. Further Reading
3. II. Data Structures
4. 2. An Array of Sequences
 - a. What's new in this chapter
 - b. Overview of Built-In Sequences
 - c. List Comprehensions and Generator Expressions
 - i. List Comprehensions and Readability
 - ii. Listcomps Versus map and filter
 - iii. Cartesian Products

iv. Generator Expressions

d. Tuples Are Not Just Immutable Lists

i. Tuples as Records

ii. Unpacking

iii. Nested Tuple Unpacking

iv. Tuples as Immutable Lists

v. Tuple versus list methods

e. Slicing

i. Why Slices and Range Exclude the Last Item

ii. Slice Objects

iii. Multidimensional Slicing and Ellipsis

iv. Assigning to Slices

f. Using + and * with Sequences

i. Building Lists of Lists

g. Augmented Assignment with Sequences

i. A += Assignment Puzzler

h. list.sort and the sorted Built-In Function

i. Managing Ordered Sequences with bisect

i. Searching with bisect

ii. Inserting with bisect.insort

j. When a List Is Not the Answer

i. Arrays

ii. Memory Views

iii. NumPy and SciPy

iv. Deques and Other Queues

k. Chapter Summary

l. Further Reading

5. 3. Dictionaries and Sets

a. What's new in this chapter

b. Standard API of Mapping Types

c. dict Comprehensions

d. Overview of Common Mapping Methods

i. Handling Missing Keys with setdefault

e. Mappings with Flexible Key Lookup

i. defaultdict: Another Take on Missing Keys

ii. The __missing__ Method

f. Variations of dict

g. Building custom mappings

i. Subclassing UserDict

h. Immutable Mappings

i. Dictionary views

j. Set Theory

i. Set Literals

ii. Set Comprehensions

iii. Set Operations

iv. Set operations on dict views

k. Internals of sets and dicts

- i. A Performance Experiment
- ii. Set hash tables under the hood
- iii. The hash table algorithm
- iv. Hash table usage in dict
- v. Key-sharing dictionary
- vi. Practical Consequences of How dict Works

l. Chapter Summary

m. Further Reading

6. 4. Text versus Bytes

- a. What's new in this chapter
- b. Character Issues
- c. Byte Essentials
- d. Basic Encoders/Decoders
- e. Understanding Encode/Decode Problems
 - i. Coping with UnicodeEncodeError
 - ii. Coping with UnicodeDecodeError
 - iii. SyntaxError When Loading Modules with Unexpected Encoding
 - iv. How to Discover the Encoding of a Byte Sequence
 - v. BOM: A Useful Gremlin
- f. Handling Text Files
 - i. Encoding Defaults: A Madhouse

- g. Normalizing Unicode for Saner Comparisons
 - i. Case Folding
 - ii. Utility Functions for Normalized Text Matching
 - iii. Extreme “Normalization”: Taking Out Diacritics
 - h. Sorting Unicode Text
 - i. Sorting with the Unicode Collation Algorithm
 - i. The Unicode Database
 - i. Finding characters by name
 - ii. Numeric meaning of characters
 - j. Dual-Mode str and bytes APIs
 - i. str Versus bytes in Regular Expressions
 - ii. str Versus bytes in os Functions
 - k. Multi-character emojis
 - i. Country flags
 - ii. Skin tones
 - iii. Rainbow flag and other ZWJ sequences
 - l. Chapter Summary
 - m. Further Reading
7. 5. Record-like data structures
- a. What’s new in this chapter
 - b. Overview of data class builders
 - i. Main features
 - c. Classic Named Tuples

- d. Typed Named Tuples
 - e. Type hints 101
 - i. No runtime effect
 - ii. Variable annotation Syntax
 - iii. The meaning of variable annotations
 - f. More about @dataclass
 - i. Field options
 - ii. Post-init processing
 - iii. Typed class attributes
 - iv. Initialization variables that are not fields
 - v. @dataclass Example: Dublin Core Resource Record
 - g. Data class as a code smell
 - i. Data class as scaffolding
 - ii. Data class as intermediate representation
 - h. Parsing binary records with struct
 - i. Structs and Memory Views
 - ii. Should we use struct?
 - i. Chapter Summary
 - j. Further Reading
8. 6. Object References, Mutability, and Recycling
- a. What's new in this chapter
 - b. Variables Are Not Boxes

- c. Identity, Equality, and Aliases
 - i. Choosing Between == and is
 - ii. The Relative Immutability of Tuples
- d. Copies Are Shallow by Default
 - i. Deep and Shallow Copies of Arbitrary Objects
- e. Function Parameters as References
 - i. Mutable Types as Parameter Defaults: Bad Idea
 - ii. Defensive Programming with Mutable Parameters
- f. del and Garbage Collection
- g. Weak References
 - i. The WeakValueDictionary Skit
 - ii. Limitations of Weak References
- h. Tricks Python Plays with Immutables
 - i. Chapter Summary
 - j. Further Reading

Part I. Prologue

Chapter 1. The Python Data Model

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at fluentpython2e@ramalho.org.

Guido’s sense of the aesthetics of language design is amazing. I’ve met many fine language designers who could build theoretically beautiful languages that no one would ever use, but Guido is one of those rare people who can build a language that is just slightly less theoretically beautiful but thereby is a joy to write programs in.¹

—Jim Hugunin, Creator of Jython, cocreator of AspectJ,
architect of the .Net DLR

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

However, if you learned another object-oriented language before Python, you may 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 describes the API that you can use to make your own objects play well with the most idiomatic

language features.

You can think of the data model as a description of Python as a framework. It formalizes the interfaces of the building blocks of the language itself, such as sequences, iterators, functions, 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. 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 (i.e., `__getitem__`). For example, the syntax `obj[key]` is supported by the `__getitem__` special method. In order to evaluate `my_collection[key]`, the interpreter calls `my_collection.__getitem__(key)`.

The special method names allow your objects to implement, support, and interact with 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 (including asynchronous context managers using `async with`).

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. Most experienced Pythonistas understand that shortcut. As a result, the special methods are also known as *dunder methods*.

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](#)”.
- [Figure 1-2](#) showing the use of special methods in “[Collection API](#)”, including the `collections.abc.Collection` abstract base class introduced in Python 3.6.

Also, here and throughout this 2nd edition I've adopted the *f-string* syntax introduced in Python 3.6, which is more readable and convenient than the older string formatting notations: the `str.format()` method and the % operator. The most common reason to use `my_fmt.format()` is to build the template `my_fmt` at runtime. That is a real need, but doesn't happen very often.

A Pythonic Card Deck

Example 1-1 is very 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. Since Python 2.6, we can 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—should be as easy as `deck[0]` or `deck[-1]`, and this is what the `__getitem__` method provides:

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



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

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



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

- 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')
...

```

The deck can also be iterated in reverse:

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

```

ELLIPSIS IN DOCTESTS

Whenever possible, the Python console listings in this book were extracted from

doctests to 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, we used the # doctest: +ELLIPSIS directive to make the doctest pass. If you are trying these examples in the interactive console, you may omit the doctest directives altogether.

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

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

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

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

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

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

```
>>> for card in sorted(deck, key=spades_high): # doctest:
+ELLIPSIS
...     print(card)
```

```
Card(rank='2', suit='clubs')
Card(rank='2', suit='diamonds')
Card(rank='2', suit='hearts')
... (46 cards omitted)
Card(rank='A', suit='diamonds')
Card(rank='A', suit='hearts')
Card(rank='A', suit='spades')
```



Although `FrenchDeck` implicitly inherits from `object`, 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 [Link to Come], 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.

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, [Link to Come] in [Link to Come].

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

Emulating Numeric Types

Several special methods allow user objects to respond to operators such as `+`. We will cover that in more detail in [Link to Come], 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](#)).

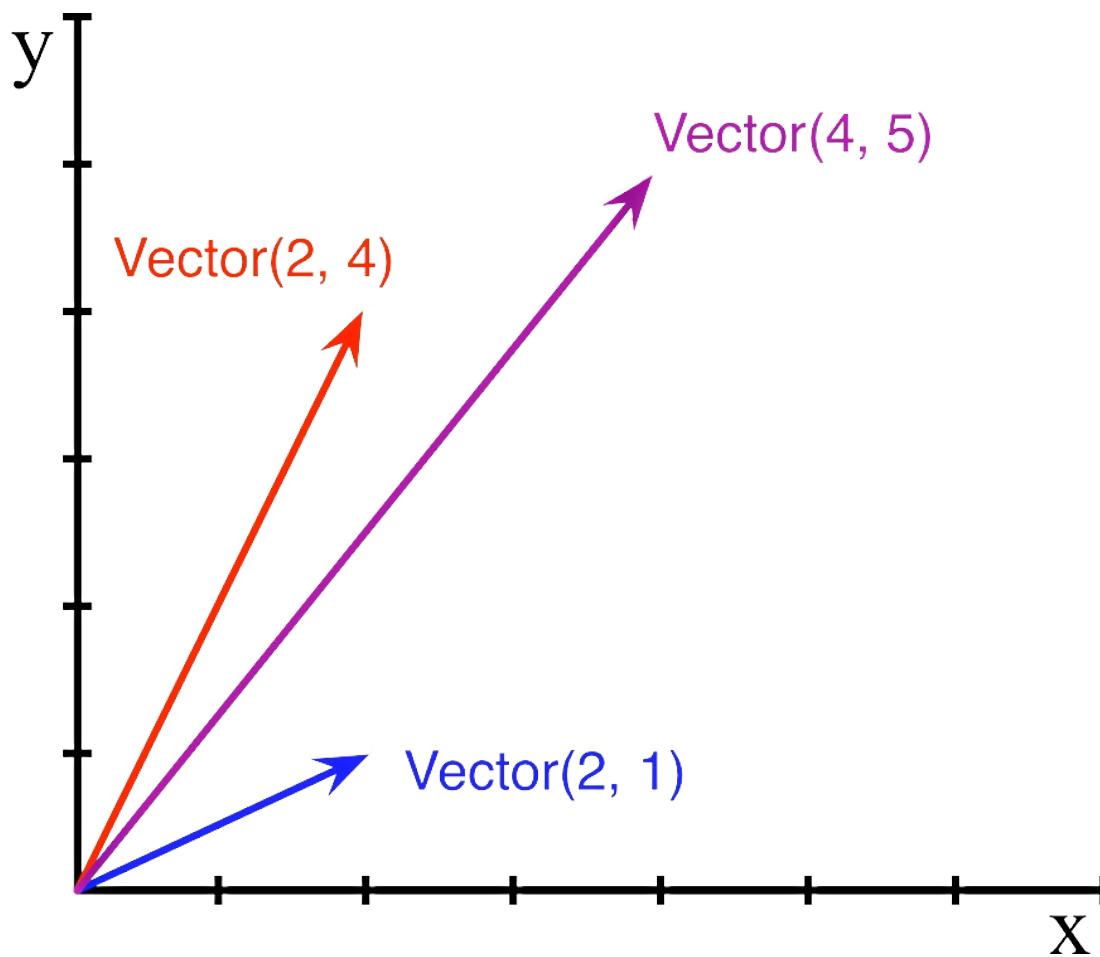


Figure 1-1. Example of two-dimensional vector addition; $\text{Vector}(2, 4) + \text{Vector}(2, 1)$ results in $\text{Vector}(4, 5)$.

TIP

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 [Link to Come].

We will start by 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 produces a `Vector` result, which is displayed in a friendly manner in the console.

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

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

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

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

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

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

```
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 six special methods, but note that none of them is directly called within the class or in the typical usage of the class illustrated by the console listings. As mentioned before, the Python interpreter is the only frequent caller of most special methods. In the following sections, we discuss the code for the 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. If we did not implement `__repr__`, vector instances would be shown in the console like `<Vector object at 0x10e100070>`.

The interactive console and debugger call `repr` on the results of the expressions evaluated, as does the `%r` 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)`).

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

If you only implement one of these special methods, choose `__repr__`, because when no custom `__str__` is available, the `__str__` method inherited from the `object` class calls `__repr__` as a fallback.

TIP

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

Arithmetic Operators

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

WARNING

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 [Link to Come].

Boolean Value of a Custom Type

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

By default, instances of user-defined classes are considered *truthy*, unless

either `__bool__` or `__len__` is implemented. Basically, `bool(x)` calls `x.__bool__()` and uses the result. If `__bool__` is not implemented, Python tries to invoke `x.__len__()`, and if that returns zero, `bool` returns `False`. Otherwise `bool` returns `True`.

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

Note how the special method `__bool__` allows your objects to be consistent with the truth value testing rules defined in the [“Built-in Types” chapter](#) of *The Python Standard Library* documentation.

NOTE

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 [Link]

to Come]. 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.

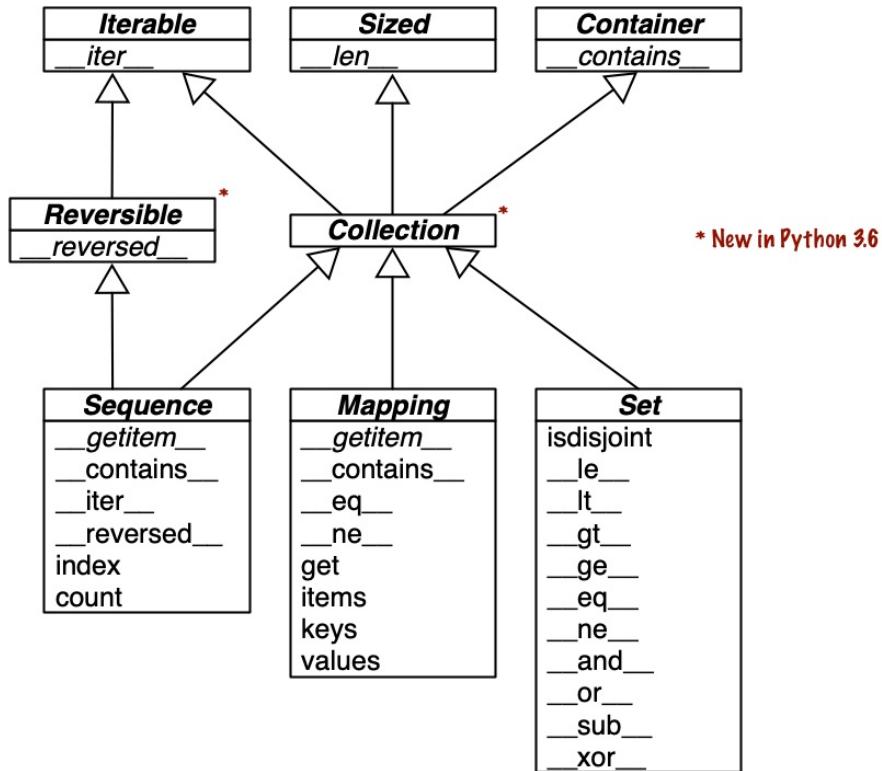


Figure 1-2. UML class diagram with fundamental collection types. Method names in italic are abstract, so they must be implemented by concrete subclasses such as `list` and `dict`. The remaining methods have concrete implementations, therefore subclasses can inherit them.

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`, comprehensions and other forms of iteration;
- `Sized` to support the `len` built-in function;
- `Contains` 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` and `collections.defaultdict`;
- `Set`: the interface of the `set` and `frozenset` built-ins.

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

NOTE

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 but one of 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 three chapters will cover built-in 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 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 in detail 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>__abs__</code> , <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>__aenter__</code> , <code>__exit__</code> , <code>__aexit__</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>
Class services	<code>__prepare__</code> , <code>__init_subclass__</code> , <code>__instancecheck__</code> , <code>__subclasscheck__</code>

Infix and numerical operators are supported by the special methods in [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 [Link to Come].

Table 1-2. Special method names for operators

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

bitwise
operators

Augmented assignment
bitwise operators `__ilshift__`, `__irshift__`, `__iand__`, `__ixor__`, `__ior__`

TIP

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

Why `len` Is Not a Method

I asked this question to core developer Raymond Hettinger in 2013 and the key to his answer was a quote from [The Zen of Python](#): “practicality beats purity.” In [“How Special Methods Are Used”](#), 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.”

NOTE

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 OO 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 widely used applications of the special methods. Making the most of sequence types is the subject of [Chapter 2](#), and implementing your own sequence will be covered in [Link to Come] 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. Implementing operators, including reversed operators and augmented assignment, will be shown in [Link to Come] 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 Edition (O’Reilly) by Alex Martelli, Anna Ravenscroft, and Steve Holden 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 Edition* (Addison-Wesley Professional), and *Python Cookbook, 3rd Edition* (O’Reilly), coauthored with Brian K. Jones.

The Art of the Metaobject Protocol (AMOP, 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 & Holden’s *Python in a Nutshell 3E*, and David Beazley’s *Python Essential Reference 4E* 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.

Magic Methods

The Ruby community calls their equivalent of the special methods *magic methods*. Many in the Python community adopt that term as well. I believe the special methods are actually the opposite of magic. Python and Ruby empower their users with a rich metaobject protocol that is not magic, enabling muggles like you and I to emulate many of the features available to core developers.

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. Less subjectively, the term *metaobject protocol* is useful to think about the Python Data Model and similar features in other languages. The *metaobject* part refers to the objects that are the building blocks of the language itself. In this context, *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 several frameworks do it, but the most important is `zope.interface`, which is briefly discussed in [Link to Come] of [Link to Come].

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

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

Part II. Data Structures

Chapter 2. An Array of Sequences

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at fluentpython2e@ramalho.org.

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.¹

—Geurts, Meertens, and Pemberton, ABC Programmer’s Handbook

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 that are new 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 [Link to Come].

What's new in this chapter

The sequence types are a very stable part of Python, therefore most the changes here are not updates but improvements over the 1st edition of *Fluent Python*. The most significant are:

- 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.
- Coverage of named tuples moved to “[Classic Named Tuples](#)” in [Chapter 5](#), where they are compared to the new data classes.

Overview of Built-In Sequences

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

Container sequences

`list`, `tuple`, and `collections.deque` can hold items of different types, including nested containers.

Flat sequences

`str`, `bytes`, `bytearray`, `memoryview`, and `array.array` hold items of one simple type.

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, and not as distinct objects. See [Figure 2-1](#).

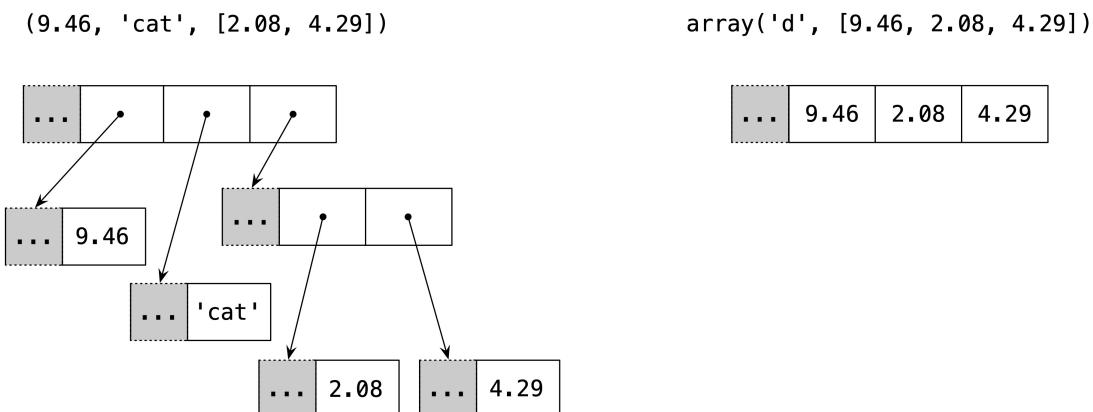


Figure 2-1. Simplified memory diagram of a tuple and an array, each holding 3 items. Gray cells represent the in-memory header of each Python object—not drawn to proportion. The tuple has an array of references to its contents. Each item is a separate Python object, possibly holding references to other Python objects, like that 2-item list. In contrast, the Python array is a single object, holding a C language array of 3 doubles.

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

NOTE

Every Python object in memory has a header with metadata. The simplest Python object, a `float`, has two metadata fields. On a 64-bit Python build, the struct representing a `float` has these 64-bit 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`. That's why an array with 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 is several objects—the tuple itself and each `float` object contained in it.

Another way of grouping sequence types is by mutability:

Mutable sequences

`list`, `bytearray`, `array.array`, `collections.deque`, and `memoryview`

Immutable sequences

`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 [Link to Come]. Being virtual subclasses, `tuple` and `list` pass these tests:

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

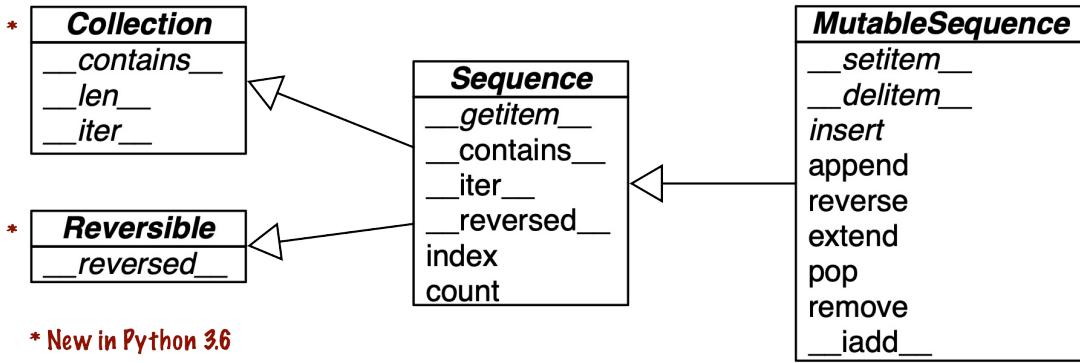


Figure 2-2. Simplified UML class diagram for some classes from `collections.abc` (superclasses are on the left; inheritance arrows point from subclasses to superclasses; names in italic are abstract classes and abstract methods)

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 them, 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 all 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.

TIP

For brevity, many Python programmers refer to list comprehensions as *listcomps*, and generator expressions as *genexp*s. 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 codepoints from a string

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

Example 2-2. Build a list of Unicode codepoints from a string, take two

```
>>> symbols = '$€¥€¤'  
>>> 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 processing tasks. The code in [Example 2-1](#) is building up a list. In contrast, a listcomp is more explicit. Its goal is 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 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, genexps, dictionaries and the like without using the ugly `\` line continuation escape. Also, when those delimiters 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 multi-line list literal, it is thoughtful to put a comma after the last item, making it a little easier for the next coder do add one more item to that list.

LOCAL SCOPE WITHIN COMPREHENSIONS AND GENERATOR EXPRESSIONS

In Python 3, list comprehensions, generator expressions, and their siblings `set` and `dict` comprehensions have their own local scope, like functions. Variables assigned within the expression are local, but variables in the surrounding scope can still be referenced. Even better, the local variables do not mask the variables from the surrounding scope.

```
>>> x = 'ABC'  
>>> codes = [ord(x) for x in x]  
>>> x ❶  
'ABC'  
>>> codes ❷  
[65, 66, 67]  
>>>
```

- x is unchanged: it's still bound to 'ABC'.
- ❶ The list comprehension produces the expected list.
- ❷ That code works, but I would not recommend using the same variable x to mean different things 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 = '$€¥€¤'  
>>> 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 [Link to Come]. 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](#).

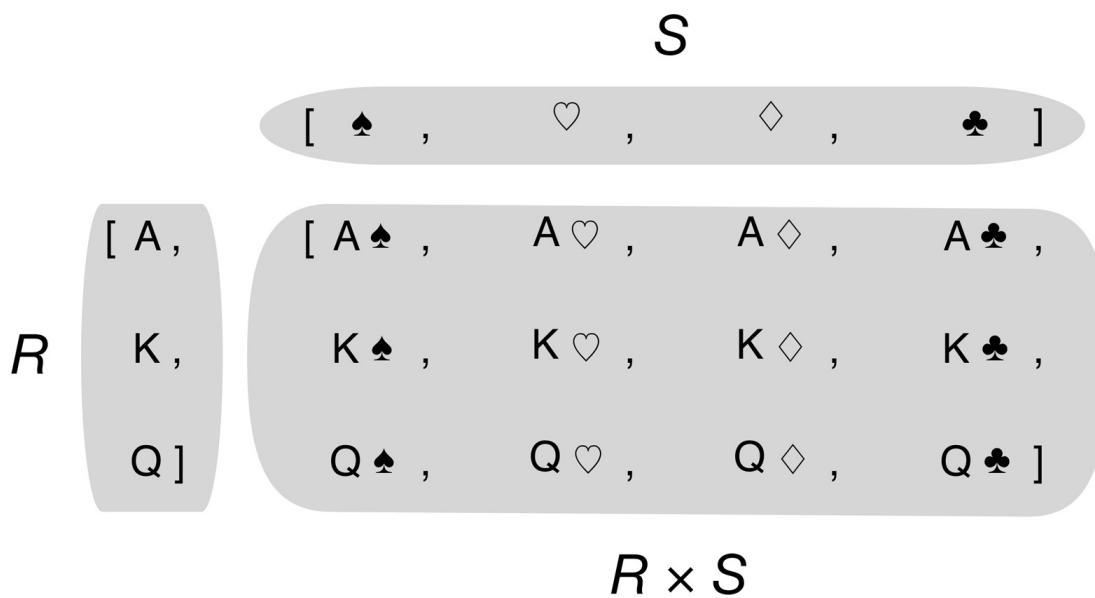


Figure 2-3. The Cartesian product of three card ranks and four suits is a sequence of twelve pairings

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 easy to see how the result will be ordered.

In [Example 1-1 \(Chapter 1\)](#), the following expression was used 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 nonlist sequences.

Generator Expressions

To initialize tuples, arrays, and other types of sequences, you could also start from a listcomp, but a genexp 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 = '$€¥¤'
>>> 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"](#)".

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 1,000 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 ('%s %s' % (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.

[Link to Come] is devoted to explaining 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 data item 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 (%), 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 it’s assigned to `_`, a dummy variable.

Tuples work well as records because of the unpacking mechanism—our next subject.

Unpacking

In [Example 2-7](#), we assigned `('Tokyo', 2003, 32_450, 0.66, 8014)` to `city`, `year`, `pop`, `chg`, `area` in a single statement. Then, in the last line, the `%` operator assigned each item in the `passport` tuple to one slot in the format string in the `print` argument. Those are two examples of *tuple unpacking*.

TIP

Tuple unpacking works with any iterable object. The only requirement is that the iterable yields exactly one item per variable in the receiving tuple, unless you use a star (*) to capture excess items as explained in “[Using * to grab excess items](#)”. 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](#).

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 tuple 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 a star 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 also shows a further use of tuple unpacking: enabling 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/idrsa.pub')
>>> filename
'idrsa.pub'
```



Sometimes when we only care about certain parts of a tuple when unpacking, a dummy variable like `_` is used as placeholder, as in the preceding example.

WARNING

If you write internationalized software, `_` is not a good dummy variable because it is

traditionally used as an alias to the `gettext.gettext` function, as recommended in the [gettext module documentation](#). Otherwise, it's a conventional name for a placeholder variable to be ignored.

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)
```

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

Nested Tuple Unpacking

The tuple to receive an expression to unpack can have nested tuples, like (a, b, (c, d)), and Python will do the right thing if the expression matches the nesting structure. [Example 2-8](#) shows nested tuple 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)),
    ('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
]

print('{:15} | {:^9} | {:^9}'.format('', 'lat.', 'long.'))
fmt = '{:15} | {:9.4f} | {:9.4f}'
for name, cc, pop, (latitude, longitude) in metro_areas: ❷
    if longitude <= 0: ❸
        print(fmt.format(name, latitude, longitude))
```

- 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 `if longitude <= 0:` limits the output to metropolitan areas in the Western hemisphere.

The output of [Example 2-8](#) is:

	lat.	long.
Mexico City	19.4333	-99.1333
New York-Newark	40.8086	-74.0204

Sao Paulo | -23.5478 | -46.6358

WARNING

Before Python 3, it was possible to define functions with nested tuples in the formal parameters (e.g., `def fn(a, (b, c), d):`). This is no longer supported in Python 3 function definitions, for practical reasons explained in [PEP 3113 — Removal of Tuple Parameter Unpacking](#). To be clear: nothing changed from the perspective of users calling a function. The restriction applies only to the definition of functions.

Python 3.5 introduced more flexible syntax for iterable unpacking, summarized in [What's New In Python 3.5](#).

As designed, tuples are very handy. But when using them as records, sometimes it is desirable to name the fields. The solution is the `namedtuple` factory we will cover in [Chapter 5](#).

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 they allow 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 demonstrates this point by creating two tuples—`a` and `b`—which are initially equal. When the last item in `b` is changed, they 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])
```

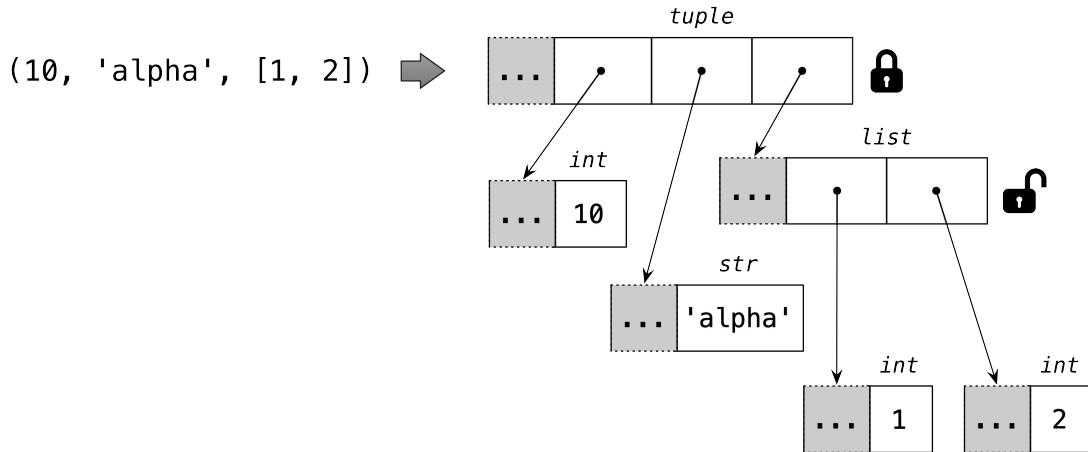


Figure 2-4. The content of the tuple itself is immutable, but that only means the references held by the tuple will always point to the same objects. However, if one of the referenced objects is a list, its content may change.

The mutable value of tuples with mutable items can be a source of bugs. If you want to make sure a `tuple` will stay unchanged, you can compute its hash. As we'll see in “[What Is Hashable?](#)”, an object is only hashable if its value cannot ever change. Therefore, here is a way to check that a tuple (or any object) has a fixed value:

```
>>> 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”](#).

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 hashable tuple `t`, the `tuple(t)` constructor just returns a reference to the same `t`. There's no need to copy, because if `t` is hashable, its value is fixed. In contrast, given a list `l`, the `list(l)` constructor must create a whole 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 within

the tuple struct itself, while a list holds a pointer to an array of references stored elsewhere. The added indirection makes CPU caches less effective, with potential impact on performance. But the indirection is necessary because when a list grows beyond the space currently allocated, the array of references needs to be relocated to make room.

Tuple versus list methods

When using a tuple as an immutable variation of list, it is good to know how similar are their APIs. 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
<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
<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 [Link to Come].

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 [Link to Come], in keeping with our philosophy of covering ready-to-use classes in this part of the book, and creating new classes in [Link to Come].

Why Slices and Range 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]
```



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

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 [Link to Come], 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 2-9](#). 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 2-9. 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 [Link to Come]. 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.²

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.³ 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, if `x` is a four-dimensional array, `x[i, ...]` is a shortcut for `x[i, :, :, :,]`. See the Tentative NumPy Tutorial 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 left 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 right side must be an iterable.

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]
>>> 5 * 'abcd'
'abcdabcdabcdabcdabcd'
```

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

WARNING

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 2-10](#).

Example 2-10. 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 2-11](#).

Example 2-11. 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
[['_', '_', 'O'], ['_', '_', 'O'], ['_', '_', 'O']]
```

- 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 2-11](#) 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 2-10](#) 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.

TIP

If either the problem or the solution in this section are 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 [Link to Come].

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
- ❸ ID of the initial tuple
- ❹ 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.⁴

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 2-12](#)?⁵

Example 2-12. 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 2-13](#) is the actual output from a Python 3.8 console.⁶

Example 2-13. 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 2-13](#).

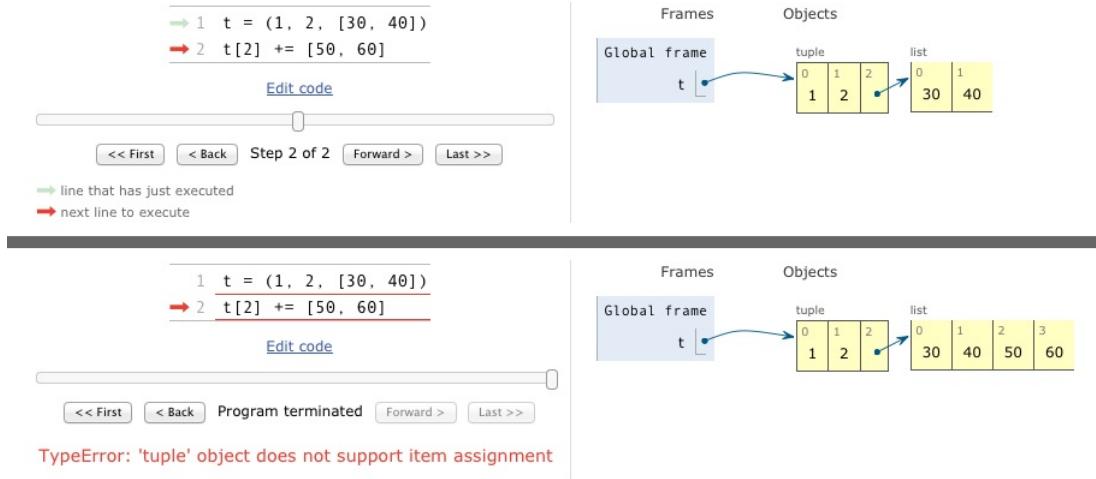


Figure 2-5. Initial and final state of the tuple assignment puzzler (diagram generated by Online Python Tutor)

If you look at the bytecode Python generates for the expression `s[a] += b` (Example 2-14), it becomes clear how that happens.

Example 2-14. 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
```

- ➊ Put the value of `s[a]` on TOS (Top Of Stack).
- ➋ Perform `TOS += b`. This succeeds if TOS refers to a mutable object (it's a list, in Example 2-13).
- ➌ Assign `s[a] = TOS`. This fails if `s` is immutable (the `t` tuple in Example 2-13).

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 and the sorted Built-In Function

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 receiver⁷ 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`.

NOTE

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.

In fact, it accepts any iterable object as an argument, including immutable sequences and generators (see [Link to Come]). 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).

TIP

The `key` optional keyword parameter can also be used 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 arguments⁸:

```
>>> 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() ❼
>>> fruits
[‘apple’, ‘banana’, ‘grape’, ‘raspberry’] ❽
```

- ❶ This produces a new list of strings sorted alphabetically⁹.
- 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 in descending order of length. 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.

WARNING

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”](#) covers proper ways of sorting text as humans would expect.

Once your sequences are sorted, they can be very efficiently searched. Fortunately, the standard binary search algorithm is already provided in the `bisect` module of the Python standard library. We discuss its

essential features next, including the convenient `bisect.insort` function, which you can use to make sure that your sorted sequences stay sorted.

Managing Ordered Sequences with `bisect`

The `bisect` module offers two main functions—`bisect` and `insort`—that use the binary search algorithm to quickly find and insert items in any sorted sequence.

Searching with `bisect`

`bisect(haystack, needle)` does a binary search for `needle` in `haystack`—which must be a sorted sequence—to locate the position where `needle` can be inserted while maintaining `haystack` in ascending order. In other words, all items appearing up to that position are less than or equal to `needle`. You could use the result of `bisect(haystack, needle)` as the `index` argument to `haystack.insert(index, needle)`—however, using `insort` does both steps, and is faster.

TIP

Raymond Hettinger wrote a [SortedCollection](#) recipe that leverages the `bisect` module and is easier to use than these standalone functions.

[Example 2-15](#) uses a carefully chosen set of “needles” to demonstrate the insert positions returned by `bisect`. Its output is in [Figure 2-6](#).

Example 2-15. `bisect` finds insertion points for items in a sorted sequence

```

import bisect
import sys

HAYSTACK = [1, 4, 5, 6, 8, 12, 15, 20, 21, 23, 23, 26, 29, 30]
NEEDLES = [0, 1, 2, 5, 8, 10, 22, 23, 29, 30, 31]

ROW_FMT = '{0:2d} @ {1:2d}     {2}{0:<2d}'

def demo(bisect_fn):
    for needle in reversed(NEEDLES):
        position = bisect_fn(HAYSTACK, needle) ❶
        offset = position * ' |' ❷
        print(ROW_FMT.format(needle, position, offset)) ❸

if __name__ == '__main__':
    if sys.argv[-1] == 'left': ❹
        bisect_fn = bisect.bisect_left
    else:
        bisect_fn = bisect.bisect

    print('DEMO:', bisect_fn.__name__) ❺
    print('haystack ->', ' '.join('%2d' % n for n in HAYSTACK))
    demo(bisect_fn)

```

- ❶ Use the chosen `bisect` function to get the insertion point.
 ❷ Build a pattern of vertical bars proportional to the `offset`.
 ❸ Print formatted row showing needle and insertion point.
 ❹ Choose the `bisect` function to use according to the last command-line argument.
 ❺ Print header with name of function selected.

```

02-array-seq/ $ python3 bisect_demo.py
DEMO: bisect
haystack ->  1  4  5  6  8  12 15  20 21 23 23 26 29 30
31 @ 14      |   |   |   |   |   |   |   |   |   |   |   |   | 31
30 @ 14      |   |   |   |   |   |   |   |   |   |   |   |   | 30
29 @ 13      |   |   |   |   |   |   |   |   |   |   |   |   | 29
23 @ 11      |   |   |   |   |   |   |   |   |   |   |   | 23
22 @ 9       |   |   |   |   |   |   |   |   | 22
10 @ 5       |   |   |   | 10
 8 @ 5       |   |   |   | 18
 5 @ 3       |   | 15
 2 @ 1       |2
 1 @ 1       |1
 0 @ 0       0

```

Figure 2-6. Output of Example 2-15 with bisect in use—each row starts with the notation needle @ position and the needle value appears again below its insertion point in the haystack

The behavior of `bisect` can be fine-tuned in two ways.

First, a pair of optional arguments, `lo` and `hi`, allow narrowing the region in the sequence to be searched when inserting. `lo` defaults to 0 and `hi` to the `len()` of the sequence.

Second, `bisect` is actually an alias for `bisect_right`, and there is a sister function called `bisect_left`. Their difference is apparent only when the needle compares equal to an item in the list: `bisect_right` returns an insertion point after the existing item, and `bisect_left` returns the position of the existing item, so insertion would occur before it. With simple types like `int`, inserting before or after makes no difference, but if the sequence contains objects that are distinct yet compare equal, then it may be relevant. For example, `1` and `1.0` are distinct, but `1 == 1.0` is `True`. Figure 2-7 shows the result of using `bisect_left`.

```

02-array-seq/ $ python3 bisect_demo.py left
DEMO: bisect_left
haystack ->  1  4  5  6  8  12 15  20 21 23 23 26 29 30
31 @ 14      |   |   |   |   |   |   |   |   |   |   |   |   | 31
30 @ 13      |   |   |   |   |   |   |   |   |   |   |   |   | 30
29 @ 12      |   |   |   |   |   |   |   |   |   |   |   |   | 29
23 @ 9       |   |   |   |   |   |   |   |   |   |   |   | 23
22 @ 9       |   |   |   |   |   |   |   |   |   |   | 22
10 @ 5        |   |   |   |   | 10
 8 @ 4        |   |   |   | 8
 5 @ 2        |   | 15
 2 @ 1        | 12
 1 @ 0        1
 0 @ 0        0

```

Figure 2-7. Output of Example 2-15 with bisect_left in use (compare with Figure 2-6 and note the insertion points for the values 1, 8, 23, 29, and 30 to the left of the same numbers in the haystack).

An interesting application of `bisect` is to perform table lookups by numeric values—for example, to convert test scores to letter grades, as in [Example 2-16](#).

Example 2-16. Given a test score, grade returns the corresponding letter grade

```

>>> def grade(score, breakpoints=[60, 70, 80, 90],
grades='FDCBA'):
...     i = bisect.bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [55, 60, 65, 70, 75, 80, 85, 90,
95]]
['F', 'D', 'D', 'C', 'C', 'B', 'B', 'A', 'A']

```

The code in [Example 2-16](#) is from the `bisect` module documentation, which also lists functions to use `bisect` as a faster replacement for the `index` method when searching through long ordered sequences of numbers.

When used for table lookups, `bisect_left` produces very different results¹⁰. Note the letter grade results in [Example 2-17](#).

Example 2-17. `bisect_left` maps a score of 60 to grade F, not D as in [Example 2-16](#).

```
>>> def grade(score, breakpoints=[60, 70, 80, 90],  
grades='FDCBA'):  
...     i = bisect.bisect_left(breakpoints, score)  
...     return grades[i]  
...  
>>> [grade(score) for score in [55, 60, 65, 70, 75, 80, 85, 90,  
95]]  
['F', 'F', 'D', 'D', 'C', 'C', 'B', 'B', 'A']
```

These functions are not only used for searching, but also for inserting items in sorted sequences, as the following section shows.

Inserting with `bisect.insort`

Sorting is expensive, so once you have a sorted sequence, it's good to keep it that way. That is why `bisect.insort` was created.

`insort(seq, item)` inserts `item` into `seq` so as to keep `seq` in ascending order. See [Example 2-18](#) and its output in [Figure 2-8](#).

Example 2-18. `Insor` keeps a sorted sequence always sorted

```
import bisect  
import random  
  
SIZE = 7  
  
random.seed(1729)  
  
my_list = []  
for i in range(SIZE):  
    new_item = random.randrange(SIZE*2)  
    bisect.insort(my_list, new_item)
```

```
print('%2d -> % new_item, my_list)
02-array-seq/ $ python3 bisect_insrt.py
10 -> [10]
 0 -> [0, 10]
  6 -> [0, 6, 10]
  8 -> [0, 6, 8, 10]
  7 -> [0, 6, 7, 8, 10]
  2 -> [0, 2, 6, 7, 8, 10]
10 -> [0, 2, 6, 7, 8, 10, 10]
```

Figure 2-8. Output of Example 2-18

Like `bisect`, `insrt` takes optional `lo`, `hi` arguments to limit the search to a sub-sequence. There is also an `insrt_left` variation that uses `bisect_left` to find insertion points.

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. If you are handling lists of numbers, arrays are the way to go. The remainder of the chapter is devoted alternatives to lists and tuples.

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 store 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 FIFO data structure.

TIP

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. But they are not sequences (their content is unordered). 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 will only contain numbers, an `array.array` is more efficient: it supports all mutable sequence operations (including `.pop`, `.insert`, and `.extend`), and 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 `signed char`. If you create an `array('b')`, then each item will be stored in a single byte and interpreted as an integer from -128 to 127. 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.

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

```
>>> from array import array ①
>>> from random import random
>>> floats = array('d', (random() for i in range(10**7))) ②
>>> floats[-1] ③
0.07802343889111107
>>> fp = open('floats.bin', 'wb')
>>> floats.tofile(fp) ④
>>> fp.close()
>>> floats2 = array('d') ⑤
>>> fp = open('floats.bin', 'rb')
>>> floats2.fromfile(fp, 10**7) ⑥
>>> fp.close()
>>> floats2[-1] ⑦
0.07802343889111107
>>> floats2 == floats ⑧
True
```

- ① Import the `array` type.
- ② Create an array of double-precision floats (typecode '`'d'`') from any iterable object—in this case, a generator expression.
- ③ Inspect the last number in the array.
- ④ Save the array to a binary file.
- ⑤ Create an empty array of doubles.
- ⑥ Read 10 million numbers from the binary file.
- ⑦ Inspect the last number in the array.
- ⑧ 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 0.1s 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 7 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.

TIP

Another fast—but more flexible—way of saving numeric data is the `pickle` module for object serialization. Saving an array of floats with `pickle.dump` is almost as fast as with `array.tofile`. However, `pickle` automatically handles almost all built-in types, including nested containers, and even instances of user-defined classes (if they are not too tricky in their implementation).

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 2-2](#), comparing the features of `list` and `array.array`.

Table 2-2. Methods and attributes found in `list` or `array` (deprecated array methods and those also implemented by `object` were 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 endianess conversion
<code>s.clear()</code>	•		Delete all items
<code>s.__contains__(e)</code>	•	•	<code>e in s</code>

)		
s.copy()	•	Shallow copy of the list
s.__copy__()	•	Support for <code>copy.copy</code>
s.count(e)	•	Count occurrences of an element
s.__deepcopy__()	•	Optimized support for <code>copy.deepcopy</code>
s.__delitem__(p)	•	Remove item at position p
s.extend(it)	•	Append items from iterable it
s.frombytes(b)	•	Append items from byte sequence interpreted as packed machine values
s.fromfile(f, n)	•	Append n items from binary file f interpreted as packed machine values
s.fromlist(l)	•	Append items from list; if one causes <code>TypeError</code> , none are appended
s.__getitem__(p)	•	s[p]—get item at position
s.index(e)	•	Find position of first occurrence of e
s.insert(p, e)	•	Insert element e before the item at position p
s.itemsize	•	Length in bytes of each array item
s.__iter__()	•	Get iterator
s.__len__()	•	<code>len(s)</code> —number of items
s.__mul__(n)	•	s * n—repeated concatenation
s.__imul__(n)	•	s *= n—in-place repeated concatenation
s.__rmul__(n)	•	n * s—reversed repeated concatenation ^a
s.pop([p])	•	Remove and return item at position p (default: last)
s.remove(e)	•	Remove first occurrence of element e 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
<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 <code>bytes</code> 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 <code>list</code>
<code>s.typecode</code>	•	One-character string identifying the C type of the items

`a` Reversed operators are explained in [Link to Come].

TIP

As of Python 3.8, the `array` type does not have an in-place `sort` method like `list.sort()`. If you need to sort an array, use the `sorted` function to rebuild it sorted:

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

To keep a sorted array sorted while adding items to it, use the `bisect.insort` function (as seen in “[Inserting with bisect.insort](#)”).

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 and SciPy](#)”). Travis Oliphant, lead author of NumPy, answers [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 2×3 matrix or a 3×2 matrix:

Example 2-20. Handling 6 bytes memory of 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 ⑦  
array('B', [0, 1, 2, 33, 22, 5])
```

- ➊ Build array of 6 bytes (typecode 'B').
- ➋ Build `memoryview` from that array, then export it as list.
- ➌ Build new `memoryview` from that previous one, but with 2 rows and 3 columns.
- ➍ Yet another `memoryview`, now with 3 rows and 2 columns.
- ➎ Ovewrite byte in `m2` at row 1, column 1 with 22.
- ➏ Ovewrite byte in `m3` at row 1, column 1 with 33.
- ➐ Display original array, proving that the memory was shared among `octets`, `m1`, `m2`, and `m3`.

Indexing a `memoryview` using a tuple—as in the expression `m2[1, 1]` above—is a feature that was added in Python 3.5.

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 an 16-bit integer array item by poking one of its bytes

```
>>> numbers = array.array('h', [-2, -1, 0, 1, 2])  
>>> memv = memoryview(numbers) ➊  
>>> len(memv)  
5  
>>> memv[0] ➋  
-2  
>>> memv_oct = memv.cast('B') ➌  
>>> memv_oct.tolist() ➍  
[254, 255, 255, 255, 0, 0, 1, 0, 2, 0]  
>>> memv_oct[5] = 4 ➋  
>>> numbers  
array('h', [-2, -1, 1024, 1, 2]) ➏
```

- ➊ 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 change to `numbers`: a 4 in the most significant byte of a 2-byte unsigned integer is 1024.

We'll see another short example with `memoryview` in the context of binary sequence manipulations with `struct` ([Chapter 4, Example 5-25](#)).

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

NumPy and SciPy

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 and SciPy are so awesome that a detour is warranted.

For advanced array and matrix operations, NumPy and SciPy are 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 elementwise 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 code base 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 demo, Example 2-22 shows some basic operations with two-dimensional arrays in NumPy.

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

```
>>> import numpy ❶
>>> a = numpy.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]) ❽
>>> a.transpose()
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).
- 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.

- ➅ Get row at index 2.
- ➆ Get element at index 2, 1.
- ➇ Get column at index 1.
- ➈ 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') ①
>>> floats[-3:] ②
array([ 3016362.69195522,    535281.10514262,   4566560.44373946])
>>> floats *= .5 ③
>>> floats[-3:]
array([ 1508181.34597761,   267640.55257131,   2283280.22186973])
>>> from time import perf_counter as pc ④
>>> t0 = pc(); floats /= 3; pc() - t0 ⑤
0.03690556302899495
>>> numpy.save('floats-10M', floats) ⑥
>>> floats2 = numpy.load('floats-10M.npy', 'r+') ⑦
>>> floats2 *= 6
>>> floats2[-3:] ⑧
memmap([ 3016362.69195522,    535281.10514262,
4566560.44373946])
```

- ➊ Load 10 million floating-point numbers from a text file.
- ➋ Use sequence slicing notation to inspect the last three numbers.
- ➌ Multiply every element in the `floats` array by .5 and inspect the last three elements again.
- ➍ Import the high-resolution performance measurement timer (available since Python 3.3).
- ➎ Divide every element by 3; the elapsed time for 10 million floats is less than 40 milliseconds.
- ➏ Save the array in a `.npy` binary file.
- ➐ 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.

- ❸ Inspect the last three elements after multiplying every element by 6.

TIP

Installing NumPy and SciPy from source is may be challenging. The [Installing the SciPy Stack](#) page on SciPy.org recommends using special scientific Python distributions such as Anaconda, Enthought Canopy, and WinPython, among others. These are large downloads, but come ready to use. Users of popular GNU/Linux distributions can usually find NumPy and SciPy in the standard package repositories. For example, you can use this command to install them on Ubuntu:

```
$ sudo apt install python-numpy python-scipy
```

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 nonnumeric 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.

Deques 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 like that, because a `deque` can be bounded—i.e., created with a fixed maximum length—and then, when it is full, it discards items from the opposite end when you add new ones. [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.

- ② 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 2-3 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 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 using locks.

Table 2-3. Methods implemented in `list` or `deque` (those that are also implemented by `object` were 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 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
<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
<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 [Link to Come].

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 `maxsize` 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 also available for easier 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 change, 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 a tuple is used as a record, tuple unpacking is the safest, most readable way of getting at the fields. The new `*` syntax makes tuple unpacking even better by making it easier to ignore some fields and to deal with optional fields. 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.

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 `key` optional argument they accept, with a function to calculate the ordering criterion. By the way, `key` can also be used with the `min` and `max` built-in functions. To keep a sorted sequence in order, always insert items into it using `bisect.insort`; to search it efficiently, use `bisect.bisect`.

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 2-3](#) and mentioning other queue implementations in the standard library.

Further Reading

Chapter 1, “Data Structures” of [*Python Cookbook, 3rd Edition*](#) (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 2-9](#).

The second edition of *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 Martelli 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 *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 left hand of parallel assignments.

If you'd like a glimpse of Python evolving, [Missing *-unpacking generalizations](#) is a bug tracker that proposed enhancements to the iterable unpacking notation. [PEP 448 — Additional Unpacking Generalizations](#) resulted from the discussions in that issue. The proposed were merged into Python 3.5, after the first edition of this book was printed.

Raymond Hettinger's response to [Are tuples more efficient than lists in Python?](#) on StackOverflow is great, but does not mention that `tuple(t)` returns `t` only if it is a hashable tuple. When `t` is unhashable, `tuple(t)` returns a shallow copy of `t`. Also, implementation details may have changed since that answer was posted in 2014. In particular, I found much smaller space savings with `sys.getsizeof` using the same example `tuple` and `list` objects as Hettinger did. Artem Golubin wrote a blog post on the same subject, titled [Optimization tricks in Python: lists and tuples](#), last updated in April, 2018 as I write this.

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 Wes McKinney's [Python for Data Analysis, 2e](#).

"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.

Fernando Perez—a physicist—created IPython, an incredibly powerful replacement for the Python console that also provides a GUI, integrated inline graph plotting, literate programming support (interleaving text with code), and rendering to PDF. Interactive, multimedia IPython sessions can be shared over HTTP as Jupyter notebooks. See screenshots and video at [The IPython Notebook](#). IPython was so successful that the project received millions of dollars in grants from the Sloan Foundation and other research institutions to hire full-time developers to enhance it. That effort resulted in [Project Jupyter](#) and the new JupyterLab web-based IDE released in July, 2019. Scientists love the combination of Python’s elegant power, Jupyter’s interactivity, and the strengths of NumPy/SciPy. That’s what made Python one of the top languages in scientific computing, paving the way for its success in the field of Machine Learning.

To learn how to use `deque` (and other collections) see the examples and practical recipes in [8.3. collections — 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 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 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 makes Python so much better and more successful than ABC. From a language implementer perspective, making tuples behave as sequences costs little. As a result, tuples may not be as "conceptually pure" as compounds, but we have many more ways of using them. They can even be used as immutable lists, of all things!

It is really useful to have immutable lists in the language, even if their type is not called `frozenlist` but is really `tuple` behaving as a sequence.

"Elegance Begets Simplicity"

The use of the syntax `*extra` to assign multiple items to a parameter started with function definitions a long time ago (I have a book about Python 1.4 from 1996 that covers that). Starting with Python 1.6, the form `*extra` can be used in the context of function calls to unpack an iterable into multiple arguments, a complementary operation. This is elegant, makes intuitive sense, and made the `apply` function redundant (it's now gone). Now, with Python 3, the `*extra` notation also works on the left of parallel assignments to grab excess items, enhancing what was already a handy language feature.

With each of these changes, Python became more flexible, more consistent, and simpler. In one word: it became more elegant. "Elegance begets simplicity" is the motto on my favorite PyCon T-shirt from Chicago, 2009. It is decorated with a painting by Bruce Eckel depicting ☰—hexagram 22 of the I Ching, (bi), "Adorning," sometimes translated as "Grace" or "Beauty."

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.”

Update: 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 unfortunate but it happened. It’s 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 capitalized initial in a single-spaced font to refer to `collections.abc.Container` or classes that implement `__contains__`.

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 is natural, considering that each item in a tuple is really a field, and each field type is independent of the others.

Key Is Brilliant

The key optional 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` actually lets us sort a mixed bag of numbers and number-like strings. You just need to decide whether you 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(1, 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 deployed 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. See [Oracle v. Google - Day 14 Filings](#).

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`.

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

2 In “[Memory Views](#)” we show that especially constructed memory views can have more than one dimension.

3 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`.

4 `str` is an exception to this description. Because string building with `+=` in loops is so common in the wild, CPython is optimized for this use case. `str` instances are allocated in memory with room to spare, so that concatenation does not require copying the whole string every time.

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

6 A reader 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.

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

8 The examples also demonstrate that Timsort—the sorting algorithm used in Python—is stable (i.e., it preserves the relative ordering of items that compare equal). Timsort is discussed further in the “Soapbox” sidebar at the end of this chapter.

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

10 Thanks to reader Gregory Sherman for pointing this out.

Chapter 3. Dictionaries and Sets

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at fluentpython2e@ramalho.org.

*May your hashes be unique,
Your keys rarely collide,
And your dictionaries
be forever ordered.¹*

—Brandon Rhodes, in *The Dictionary Even Mightier*

The `dict` type is not only widely used in our programs but also a fundamental part of the Python implementation. Class and instance attributes, module namespaces, and function keyword arguments are some of the fundamental Python constructs represented by dictionaries in memory. The built-in functions are all in `__builtins__.__dict__`.

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:

- Common dictionary methods
- Special handling for missing keys
- Variations of `dict` in the standard library
- The `set` and `frozenset` types
- How hash tables work
- Implications of hash tables in the behavior of sets and dictionaries.

What's new in this chapter

The `dict` implementation evolved from what I described in 1st edition of *Fluent Python*. Major revisions in this chapter are:

- Explanation of the hash table algorithm now starts with its use in `set`, which is simpler to understand.
- Coverage of the memory optimizations that preserve key insertion order in `dict` instances—implemented in Python 3.6—and the key-sharing layout for dictionaries holding instance attributes—the `__dict__` of user-defined objects since Python 3.3.
- New section on the view objects returned by `dict.keys`, `dict.items`, and `dict.values` since Python 3.0.

NOTE

One minor change: I adopted the term *hash code* instead of *hash value*, because we need to talk about object values to understand hashing, and it is easier to keep the two concepts apart in a sentence like this: “The *hash code* is derived from the *object value*.” However, I keep the original term when I quote the Python documentation.

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](#).

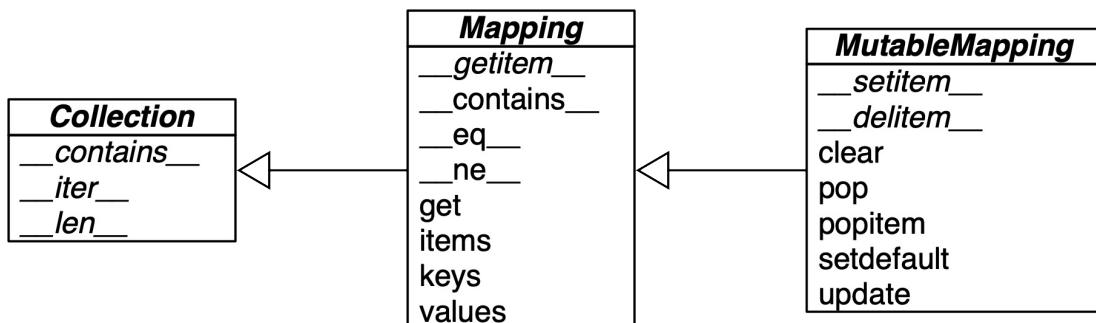


Figure 3-1. Simplified UML class diagram for the `MutableMapping` and its superclasses from `collections.abc` (inheritance arrows point from subclasses to superclasses; names in italic are abstract classes and abstract methods)

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
```

TIP

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 [Link to Come].

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, check out "[What Is Hashable?](#)".

WHAT IS HASHABLE?

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

An object is hashable if it has a hash value 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 value. [...]

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 by definition. A `tuple` is hashable only if all its items are hashable. See tuples `tt`, `t1`, and `tf`:

```
>>> tt = (1, 2, (30, 40))
>>> hash(tt)
8027212646858338501
>>> t1 = (1, 2, [30, 40])
>>> hash(t1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> tf = (1, 2, frozenset([30, 40]))
>>> hash(tf)
-4118419923444501110
```



User-defined types are hashable by default because their `id()` and the `__eq__()` method inherited from the `object` class simply compares the object ids. If an object implements a custom `__eq__()` which 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.

Given these ground rules, you can build dictionaries in several ways. The [Built-in Types](#) page in the Library Reference has this example to show the various means of building a dict:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'three': 3, 'two': 2, 'one': 1}
>>> c = dict([('two', 2), ('one', 1), ('three', 3)])
>>> d = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

Note that all of those `dict` instances are considered equal because they have the same set of keys and values, even if the order of the keys is not the same.

C_Python 3.6 started preserving the insertion order of the keys as an implementation detail, and Guido van Rossum declared it an offical language feature in Python 3.7, so we can depend on it:

```
>>> a
{'one': 1, 'two': 2, 'three': 3}
>>> list(a.keys())
['one', 'two', 'three']
>>> c
{'two': 2, 'one': 1, 'three': 3}
>>> c.popitem()
('three', 3)
>>> c
{'two': 2, 'one': 1}
```

Before Python 3.6, `c.popitem()` would remove and return an arbitrary key-value pair. Now it always removes and returns the last key-value pair added to the `dict`.

In addition to the literal syntax and the flexible `dict` constructor, we can use *dict comprehensions* to build dictionaries. See the next section.

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* 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 = [
❶    ...
    (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}
❷
>>> 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()}
```

```

③
...
    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.s
- ❸ Sorting `country_dial` by name, reversing the pairs again, uppercasing values, and filtering items by `code < 66`.

If you're used to liscomps, 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.

We now move to a panoramic view of the API for mappings.

Overview of Common Mapping Methods

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

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</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 k
<code>d.fromkeys(it, [initial])</code>	•	•	•	New mapping from keys in iterable, with optional initial value (defaults to None)
<code>d.get(k, [default])</code>	•	•	•	Get item with key k, return default or None if missing
<code>d.__getitem__(k)</code>	•	•	•	<code>d[k]</code> —get item with key k
<code>d.items()</code>	•	•	•	Get view over items—(key, value) pairs
<code>d.__iter__()</code>	•	•	•	Get iterator over keys
<code>d.keys()</code>	•	•	•	Get view 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 k first or last position (last is True by default)
<code>d.pop(k, [default])</code>	•	•	•	Remove and return value at k, or default or None if missing
<code>d.popitem()</code>	•	•	•	Remove and return the last inserted item as (ke

			y, value)	^b
d.__reversed__ _()	•	•	•	Get iterator for keys from last to first inserted
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, [**kargs])	•	•	•	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). This keyword argument is not valid for `dict` or `defaultdict` as of Python 3.8.

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 the value of a dictionary item is mutable and we need to update it in-place. If you are not comfortable using it, the following section explains how, through a practical example.

Handling Missing Keys with `setdefault`

In line with Python's *fail-fast* philosophy, `dict` access with `d[k]` raises an error when `k` is not an existing key. Every Pythonista knows that `d.get(k, default)` is an alternative to `d[k]` whenever a default value is more convenient than handling `KeyError`. However, when updating the mutable value found, using either `d[k]` or `get` is awkward and inefficient.

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 3-2](#).

Example 3-2. Partial output from Example 3-3 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 3-3](#), 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,².

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

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

```
import sys
import re

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

# print 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`.
- ❹ 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.³

The three lines dealing with `occurrences` in Example 3-3 can be replaced by a single line using `dict.setdefault`. Example 3-4 is closer to Alex Martelli's original example.

Example 3-4. 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-

3

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


```

```
import sys
import re

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

# print 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.

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

Mappings with Flexible Key Lookup

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

[Example 3-5](#) uses `collections.defaultdict` to provide another elegant solution to the problem in [Example 3-4](#). A `defaultdict` is configured to create items on demand whenever a missing key is searched.

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

For example, given an empty `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 called `default_factory`.

Example 3-5. index_default.py: using an instance of defaultdict instead of the setdefault method

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


```

```
import sys
import re
import collections

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

# print 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.

WARNING

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`.

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`.

WARNING

The `__missing__` method is only called by `__getitem__` (i.e., for the `d[k]` operator). The presence of a `__missing__` method has no effect on the behavior of other methods that look up keys, such as `get` or `__contains__` (which implements the `in` operator). This is why the `default_factory` of `defaultdict` works only with `__getitem__`, as noted in the warning at the end of the previous section.

Suppose you'd like a mapping where keys are converted to `str` when looked up. A concrete use case is a device library for IoT⁴, 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 that 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 3-6 shows how such a mapping would work.

Example 3-6. 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 3-7 implements a class `StrKeyDict0` that passes the preceding doctests.

TIP

A better way to create a user-defined mapping type is to subclass `collections.UserDict` instead of `dict` (as we'll do in [Example 3-8](#)). Here we subclass `dict` just to show that `__missing__` is supported by the built-in `dict.__getitem__` method.

Example 3-7. StrKeyDict0 converts nonstring keys to str on lookup (see tests in Example 3-6)

```
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. The last line, `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()`.

NOTE

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](#)”. 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 3-7](#).

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.

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, we summarize the various mapping types included in the standard library, besides `defaultdict`, already covered in [“`defaultdict`: Another Take on Missing Keys”](#).

The following mapping types are ready to instantiate and use:

`collections.OrderedDict`

Maintains keys in insertion order, allowing iteration over items in a predictable order. The `popitem` method of an `OrderedDict` pops the last item by default, but if called as `my_odict.popitem(last=False)`, it pops the first item added. Now that the built-in `dict` also keeps the keys ordered since Python 3.6, the main reason to use `OrderedDict` is writing code that is backward-compatible with earlier Python versions.

`collections.ChainMap`

Holds a list of mappings that can be searched as one. The lookup is performed on each mapping in order, and succeeds if the key is found in any of them. This is useful to interpreters for languages with nested scopes, where each mapping represents a scope context. 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))
```



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 (see below). `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('aaaaazzz')
>>> 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, each element in the set is a key, and the count is the number of occurrences of that element in the set.

`OrderedDict`, `ChainMap`, and `Counter` are ready to instantiate but can also be customized by subclassing. In contrast, the next mappings are intended as base classes to be extended.

Building custom mappings

These mapping types are not meant to be instantiated directly, but for subclassing when we need to create custom types:

collections.UserDict

A pure Python implementation of a mapping that behaves like a standard `dict`. See “[Subclassing UserDict](#)” for an extended explanation.

typing.TypedDict

Using new type declaration syntax, the `TypedDict` class added in Python 3.8 lets you create mapping types that only accept a specific set of string keys, with each key constrained to accept only values of a specific type. This is covered in [Link to Come], in [Chapter 5](#).

The `collections.UserDict` class behaves like a `dict`, but it is slower because it is implemented in Python, not in C. We’ll cover it in more detail next.

Subclassing UserDict

It’s almost always easier to create a new mapping type by extending `UserDict` rather than `dict`. We realize that when we try to extend our `StrKeyDict0` from [Example 3-7](#) 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.⁵

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 3-7](#).

Thanks to `UserDict`, `StrKeyDict` ([Example 3-8](#)) is actually shorter than `StrKeyDict0` ([Example 3-7](#)), 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 3-8. StrKeyDict always converts non-string 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 3-7](#).
- 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.

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 3-7](#)), we had to code our own `get` to obtain results consistent with `__getitem__`, but in [Example 3-8](#) we inherited `Mapping.get`, which is implemented exactly like `StrKeyDict0.get` (see [Python source code](#)).

TIP

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 standalone module (`03-dict-set/transformdict.py` in the [Fluent Python 2nd 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. Read on.

Immutable Mappings

The mapping types provided by the standard library are all mutable, but you may need to guarantee that a user cannot change a mapping by

mistake. A concrete use case can be found, again, in a hardware programming library “[The missing Method](#)”: the `board.pins` mapping represents the physical GPIO pins on the device. As such, it’s nice 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.

Since Python 3.3, 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 3-9](#) for a brief demonstration.

Example 3-9. `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 one of the most powerful features of dictionaries in Python 3: `dict` views.

Dictionary views

The `dict` instance methods `.keys()`, `.values()`, and `.items()` return instances of classes called `dict_keys`, `dict_values`, and `dict_items`, respectively⁶. 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 3-10 shows some basic operations supported by all dictionary views.

Example 3-10. 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) ❸
<dict_reversevalueiterator object at 0x10e9e7310>
>>> values[0] ❹
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 3-10](#):

```
>>> 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](#)”.

Now that we've covered most mapping types in the standard library, we'll review 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.

NOTE

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']
```

TIP

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 3-11](#)).

Example 3-11. Count occurrences of needles in a haystack, both of type set

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

Without the intersection operator, you'd have write [Example 3-12](#) to accomplish the same task as [Example 3-11](#).

Example 3-12. Count occurrences of needles in a haystack (same end result as Example 3-11)

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



[Example 3-11](#) runs slightly faster than [Example 3-12](#). On the other hand, [Example 3-12](#) works for any iterable objects `needles` and `haystack`, while [Example 3-11](#) 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 3-13](#).

Example 3-13. 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 3-13](#), but if either the `needles` or the `haystack` is already a set, the alternatives in [Example 3-13](#) may be cheaper than [Example 3-12](#).

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: 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` bytecode⁷.

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 in Python 2.7, together with the dictcomps that we saw in “[dict Comprehensions](#)”. [Example 3-14](#) shows how.

Example 3-14. Build a set of Latin-1 characters that have the word “SIGN” in their Unicode names

```
>>> from unicodedata import name ①
>>> {chr(i) for i in range(32, 256) if 'SIGN' in name(chr(i), '')}
②
{'$', '=', '¢', '#', '¤', '<', '¥', 'µ', '×', '$', '¶', '£', '©',
 '°', '+', '÷', '±', '>', '¬', '®', '%'}
```

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

Syntax matters aside, 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 \geq . 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`.

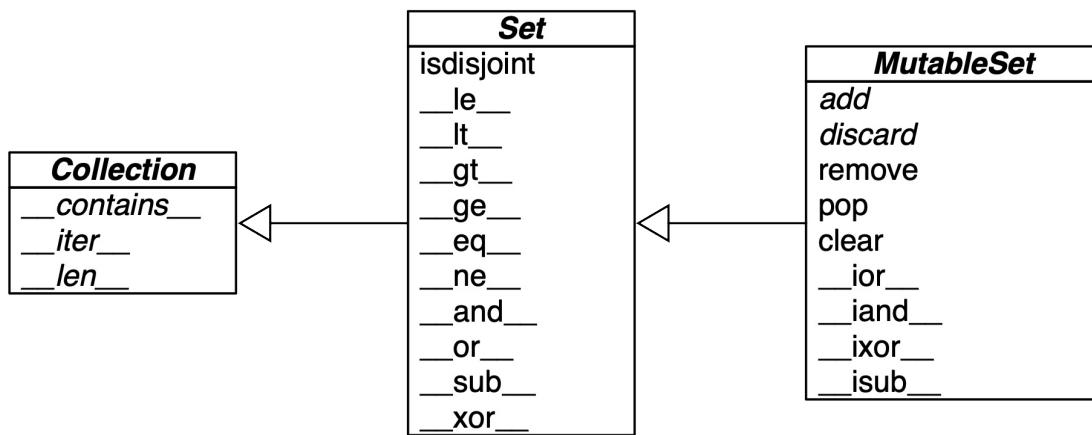


Figure 3-2. Simplified UML class diagram for `MutableSet` and its superclasses from `collections.abc` (names in italic are abstract classes and abstract methods; reverse operator methods omitted for brevity)

TIP

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.

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	Python
------	--------

symbol	operator	Method	Description
$S \cap Z$	$s \& z$	<code>s.__and__(z)</code>	Intersection of <code>s</code> and <code>z</code>
	$z \& s$	<code>s.__rand__(z)</code>	Reversed & operator
		<code>s.intersection(it, ...)</code>	Intersection of <code>s</code> and all sets built from iterables <code>it</code> , etc.
	$s \&= z$	<code>s.__iand__(z)</code>	<code>s</code> updated with intersection of <code>s</code> and <code>z</code>
		<code>s.intersection_update(it, ...)</code>	<code>s</code> updated with intersection of <code>s</code> and all sets built from iterables <code>it</code> , etc.
$S \cup Z$	$s z$	<code>s.__or__(z)</code>	Union of <code>s</code> and <code>z</code>
	$z s$	<code>s.__ror__(z)</code>	Reversed
		<code>s.union(it, ...)</code>	Union of <code>s</code> and all sets built from iterables <code>it</code> , etc.
	$s = z$	<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.
$S \setminus Z$	$s - z$	<code>s.__sub__(z)</code>	Relative complement or difference between <code>s</code> and <code>z</code>
	$z - s$	<code>s.__rsub__(z)</code>	Reversed - operator
		<code>s.difference(it, ...)</code>	Difference between <code>s</code> and all sets built from iterables <code>it</code> , etc.
	$s -= z$	<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.
		<code>s.symmetric_difference(it)</code>	Complement of <code>s & set(it)</code>

$S \Delta Z$	<code>s ^ z</code>	<code>s.__xor__(z)</code>	Symmetric difference (the complement of the intersection $S \cap Z$)
$Z \wedge S$	<code>s.__rxor__(z)</code>		Reversed \wedge operator
		<code>s.symmetric_difference_update(it, ...)</code>	S updated with symmetric difference of S and all sets built from iterables <code>it</code> , etc.
$S \wedge= Z$	<code>s.__ixor__(z)</code>		S updated with symmetric difference of S and Z

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
		<code>s.isdisjoint(z)</code>	S and Z are disjoint (no elements in common)
$e \in S$	<code>e in s</code>	<code>s.__contains__(e)</code>	Element e is a member of S
$S \subseteq Z$	<code>s <= z</code>	<code>s.__le__(z)</code>	S is a subset of the Z set
		<code>s.issubset(it)</code>	S 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>	S is a proper subset of the Z set
		<code>s.__ge__(z)</code>	S is a superset of the Z set

	<code>s.issuperset (it)</code>	<code>s</code> is a superset of the set built from the iterable <code>it</code>
<code>S ⊃ Z</code>	<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

	<code>set</code>	<code>frozenset</code>	
<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</code> not in <code>s</code>

This completes our overview of the features of sets. As promised in “Dictionary views”, 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 & 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>)
<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__(z)</code>	•	•	•	<code>s ^ z</code> (symmetric difference of <code>s</code> and <code>z</code>)
<code>s.__rxor__(z)</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).

This means, for example, that finding the keys that appear in two dictionaries is as easy as this:

```
>>> 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'}
```



This will save a lot of loops and ifs when inspecting the contents of dictionaries in your code.

We now change gears to discuss how sets and dictionaries are implemented with hash tables. After reading the rest of this chapter, you should no longer be surprised by the behavior of `dict`, `set`, and other data structures powered by hash tables.

Internals of sets and dicts

Understanding how Python dictionaries and sets are built with hash tables is helpful to make sense of their strengths and limitations.

NOTE

Consider this section optional. You don't need to know all of these details to make good use of dictionaries and sets. But the implementation ideas are beautiful—that's why I covered them. For practical advice, you can skip to “[Practical Consequences of How Sets Work](#)” and “[Practical Consequences of How dict Works](#)”.

Here are some questions this section will answer:

- How efficient are Python `dict` and `set`?
- Why are `set` elements unordered?
- Why can't we use any Python object as a `dict` key or `set` element?
- Why does the order of the `dict` keys depend on insertion order?
- Why does the order of `set` elements seem random?

To motivate the study of hash tables, we start by showcasing the amazing performance of `dict` and `set` with a simple test involving millions of items.

A Performance Experiment

From experience, all Pythonistas know that dicts and sets are fast. We'll confirm that with a controlled experiment.

To see how the size of a `dict`, `set`, or `list` affects the performance of search using the `in` operator, I generated an array of 10 million distinct double-precision floats, the "haystack." I then generated an array of needles: 1,000 floats, with 500 picked from the haystack and 500 verified not to be in it.

For the `dict` benchmark, I used `dict.fromkeys()` to create a `dict` named `haystack` with 1,000 floats. This was the setup for the `dict` test. The actual code I clocked with the `timeit` module is [Example 3-15](#) (like [Example 3-12](#)).

Example 3-15. Search for needles in haystack and count those found

```
found = 0
```

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

I repeated the benchmark five times, each time increasing tenfold the size of `haystack`, from 1,000 to 10,000,000 items. The result of the `dict` performance test is in [Table 3-6](#).

Table 3-6. Total time for using `in` operator to search for 1,000 needles in haystack dicts of five sizes on a 2.2 GHz Core i7 laptop running Python 3.8.0 (tests timed the loop in Example 3-15)

len of haystack	Factor	dict time	Factor
1,000	1×	0.099ms	1.00×
10,000	10×	0.109ms	1.10×
100,000	100×	0.156ms	1.58×
1,000,000	1,000×	0.372ms	3.76×
10,000,000	10,000×	0.512ms	5.17×

In concrete terms, to check for the presence of 1,000 floating-point keys in a dictionary with 1,000 items, the processing time on my laptop was 99μs, and the same search in a `dict` with 10,000,000 items took 512μs. In other words, the average time for each search in the haystack with 10 million items was 0.512μs—yes, that’s about half microsecond per needle. When the search space became 10,000 times larger, the search time increased a little over 5 times. Nice.

To compare with other collections, I repeated the benchmark with the same haystacks of increasing size, but storing the `haystack` as a `set` or as `list`. For the `set` tests, in addition to timing the `for` loop in

Example 3-15, I also timed the one-liner in Example 3-16, which produces the same result: count the number of elements from `needles` that are also in `haystack`—if both are sets.

Example 3-16. Use set intersection to count the needles that occur in haystack

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



Table 3-7 shows the tests side by side. The best times are in the “set& time” column, which displays results for the set & operator using the code from Example 3-16. As expected, the worst times are in the “list time” column, because there is no hash table to support searches with the `in` operator on a `list`, so a full scan must be made if the needle is not present, resulting in times that grow linearly with the size of the haystack.

Table 3-7. Total time for using `in` operator to search for 1,000 keys in haystacks of 5 sizes, stored as dicts, sets, and lists on a 2.2 GHz Core i7 laptop running Python 3.8.0 (tests timed the loop in Example 3-15 except the `set&`, which uses Example 3-16)

len of haystack	Factor	dict time	Factor	set time	Factor	set& time	Factor	list time
1,000	1×	0.099ms	1.00×	0.107ms	1.00×	0.083ms	1.00×	9.115ms
10,000	10×	0.109ms	1.10×	0.119ms	1.11×	0.094ms	1.13×	78.219ms
100,000	100×	0.156ms	1.58×	0.147ms	1.37×	0.122ms	1.47×	767.971μs
1,000,000	1,000×	0.372ms	3.76×	0.264ms	2.47×	0.240ms	2.89×	8,020.3μs
10,000,000	10,000×	0.512ms	5.17×	0.330ms	3.08×	0.298ms	3.59×	78,558.μs

If your program does any kind of I/O, the lookup time for keys in dicts or sets is negligible, regardless of the `dict` or `set` size (as long as it fits in

RAM). See the code used to generate [Table 3-7](#) and accompanying discussion in [Link to Come], [Link to Come].

Now that we have concrete evidence of the speed of dicts and sets, let's explore how that is achieved with the help of hash tables.

Set hash tables under the hood

Hash tables are a wonderful invention. Let's see how a hash table is used when adding elements to a set.

Let's say we have a set with abbreviated workdays, created like this:

```
>>> workdays = {'Mon', 'Tue', 'Wed', 'Thu', 'Fri'}  
>>> workdays  
{'Tue', 'Mon', 'Wed', 'Fri', 'Thu'}
```

The core data structure of a Python `set` is a hash table with at least 8 rows. Traditionally, the rows in hash table are called *buckets*⁸.

A hash table holding the elements of `workdays` looks like [Figure 3-3](#).

hash code	pointer to element
0	-1
1	-1
2	2414279730484651250 → 'Tue'
3	4199492796428269555 → 'Mon'
4	-5145319347887138165 → 'Wed'
5	7021641685991143771 → 'Fri'
6	-1
7	-1139383146578602409 → 'Thu'

Figure 3-3. Hash table for the set {'Mon', 'Tue', 'Wed', 'Thu', 'Fri'}. Each bucket has two fields: the hash code and a pointer to the element value. Empty buckets have -1 in the hash code field. The ordering looks random.

In CPython built for a 64-bit CPU, each bucket in a set has two fields: a 64-bit hash code, and a 64-bit pointer to the element value—which is a Python object stored elsewhere in memory. Because buckets have a fixed size, access to an individual bucket is done by offset. There is no field for the indexes from 0 to 7 in [Figure 3-3](#).

Before covering the hash table algorithm, we need to know more about hash codes, and how they relate to equality.

HASHES AND EQUALITY

The `hash()` built-in function works directly with built-in types and falls back to calling `__hash__` for user-defined types. If two objects compare equal, their hash codes must also be equal, otherwise the hash table algorithm does not work. For example, because `1 == 1.0` is `True`, `hash(1) == hash(1.0)` must also be `True`, even though the internal representation of an `int` and a `float` are very different.⁹

Also, to be effective as hash table indexes, hash codes should scatter around the index space as much as possible. This means that, ideally, objects that are similar but not equal should have hash codes that differ widely. [Example 3-17](#) is the output of a script to compare the bit patterns of hash codes. Note how the hashes of 1 and 1.0 are the same, but those of 1.0001, 1.0002, and 1.0003 are very different.

Example 3-17. Comparing hash bit patterns of 1, 1.0001, 1.0002, and 1.0003 on a 32-bit build of Python (bits that are different in the hashes above and below are highlighted with ! and the right column shows the number of bits that differ)



The code to produce [Example 3-17](#) is in [\[Link to Come\]](#). Most of it deals with formatting the output, but it is listed as [\[Link to Come\]](#) for completeness.

NOTE

Starting with Python 3.3, a random salt value is included when computing hash codes for `str`, `bytes`, and `datetime` objects, as documented in [Issue 13703—Hash collision security issue](#). The salt value is constant within a Python process but varies between interpreter runs. With PEP-456, Python 3.4 adopted the SipHash cryptographic function to compute hash codes for `str` and `bytes` objects. The random salt and SipHash are security measures to prevent DoS attacks. Details are in a note in the documentation for the `__hash__` special method.

HASH COLLISIONS

As mentioned, on 64-bit CPython a hash code is a 64-bit number, and that's 2^{64} possible values—which is more than 10^{19} . But most Python types can represent many more different values. For example, a string made of 10 ASCII printable characters picked at random has 100^{10}

possible values—more than 2^{64} . Therefore, the hash code of an object usually has less information than the actual object value. This means that objects that are different may have the same hash code.

TIP

When correctly implemented, hashing guarantees that different hash codes always imply different objects, but the reverse is not true: different objects don't always have different hash codes. When different objects have the same hash code, that's a *hash collision*.

With this basic understanding of hash codes and object equality, we are ready to dive into the algorithm that makes hash tables work, and how hash collisions are handled.

The hash table algorithm

We will focus on the internals of `set` first, and later transfer the concepts to `dict`.

NOTE

This is a simplified view of how Python uses a hash table to implement a `set`. For all details, see commented source code for CPython's `set` and `frozenset` in [Include/setobject.h](#) and [Objects/setobject.c](#).

Let's see how Python builds a set like `{'Mon', 'Tue', 'Wed', 'Thu', 'Fri'}`, step by step. The algorithm is illustrated by the flowchart in [Figure 3-4](#), and described next.

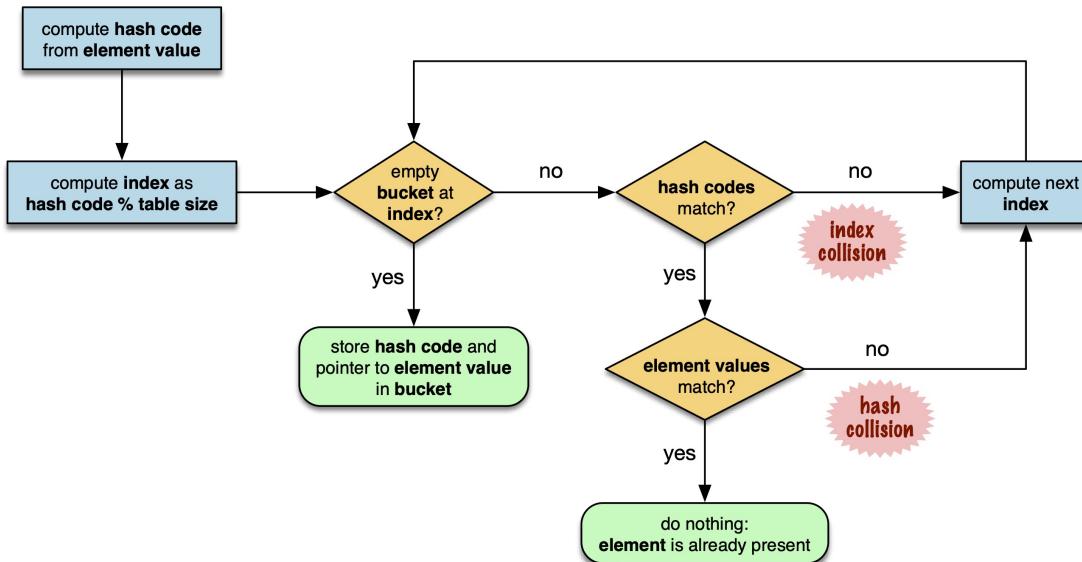


Figure 3-4. Flowchart for algorithm to add element to the hash table of a set.

Step 0: initialize hash table

As mentioned earlier, the hash table for a `set` starts with 8 empty buckets. As elements are added, Python makes sure at least $\frac{1}{3}$ of the buckets are empty—doubling the size of the hash table when more space is needed. The hash code field of each bucket is initialized with -1, which means “no hash code”¹⁰.

Step 1: compute the hash code for the element

Given the literal `{'Mon', 'Tue', 'Wed', 'Thu', 'Fri'}`, Python gets the hash code for the first element, `'Mon'`. For example, here is a realistic hash code for `'Mon'`—you’ll probably get a different result because of the random salt Python uses to compute the hash code of strings:

```
>>> hash('Mon')
4199492796428269555
```

Step 2: probe hash table at index derived from hash code

Python takes the modulus of the hash code with the table size to find a hash table index. Here the table size is 8, and the modulus is 3:

```
>>> 4199492796428269555 % 8  
3
```

Probing consists of computing the index from the hash, then looking at the corresponding bucket in the hash table. In this case, Python looks at the bucket at offset 3 and finds -1 in the hash code field, marking an empty bucket.

Step 3: put the element in the empty bucket

Python stores the hash code of the new element, 4199492796428269555, in the hash code field at offset 3, and a pointer to the string object 'Mon' in the element field. Figure 3-5 shows the current state of the hash table.

	hash code	pointer to element
0	-1	
1	-1	
2	-1	
3	4199492796428269555	→ 'Mon'
4	-1	
5	-1	
6	-1	
7	-1	

Figure 3-5. Hash table for the set { 'Mon' }.

STEPS FOR REMAINING ITEMS

For the second element, 'Tue', steps 1, 2, 3 above are repeated. The hash code for 'Tue' is 2414279730484651250, and the resulting index is 2.

```

>>> hash('Tue')
2414279730484651250
>>> hash('Tue') % 8
2

```

The hash and pointer to element 'Tue' are placed in bucket 2, which was also empty. Now we have [Figure 3-6](#)

hash code	pointer to element
0	-1
1	-1
2	2414279730484651250 → 'Tue'
3	4199492796428269555 → 'Mon'
4	-1
5	-1
6	-1
7	-1

Figure 3-6. Hash table for the set {'Mon', 'Tue'}. Note that element ordering is not preserved in the hash table.

STEPS FOR A COLLISION

When adding 'Wed' to the set, Python computes the hash -5145319347887138165 and index 3. Python probes bucket 3 and sees that it is already taken. But the hash code stored there, 4199492796428269555 is different. As discussed in "[Hashes and equality](#)", if two objects have different hashes, then their value is also different. This is an index collision. Python then probes the next bucket and finds it empty. So 'Wed' ends up at index 4, as shown in [Figure 3-7](#).

hash code	pointer to element
0	-1
1	-1
2	2414279730484651250 → 'Tue'
3	4199492796428269555 → 'Mon'
4	-5145319347887138165 → 'Wed'
5	-1
6	-1
7	-1

Figure 3-7. Hash table for the set `{'Mon', 'Tue', 'Wed'}`. After the collision, 'Wed' is put at index 4.

Adding the next element, 'Thu', is boring: there's no collision, and it lands in its natural bucket, at index 7.

Placing 'Fri' is more interesting. Its hash, 7021641685991143771 implies index 3, which is taken by 'Mon'. Probing the next bucket—4—Python finds the hash for 'Wed' stored there. The hash codes don't match, so this is another index collision. Python probes the next bucket. It's empty, so 'Fri' ends up at index 5. The end state of the hash table is shown in [Figure 3-8](#).

NOTE

Incrementing the index after a collision is called *linear probing*. This can lead to clusters of occupied buckets, which can degrade the hash table performance, so CPython counts the number of linear probes and after a certain threshold, applies a pseudo random number generator to obtain a different index from other bits of the hash code. This optimization is particularly important in large sets.

	hash code	pointer to element
0	-1	
1	-1	
2	2414279730484651250	————→ 'Tue'
3	4199492796428269555	————→ 'Mon'
4	-5145319347887138165	————→ 'Wed'
5	7021641685991143771	————→ 'Fri'
6	-1	
7	-1139383146578602409	————→ 'Thu'

Figure 3-8. Hash table for the set `{'Mon', 'Tue', 'Wed', 'Thu', 'Fri'}`. It is now 62.5% full—close to the $\frac{3}{4}$ threshold.

When there is an element in the probed bucket and the hash codes match, Python also needs to compare the actual object values. That's because, as explained in “[Hash collisions](#)”, it's possible that two different objects have the same hash code—although that's rare for strings, thanks to the quality of the Siphash algorithm¹¹. This explains why hashable objects must implement both `__hash__` and `__eq__`.

If a new element were added to our example hash table, it would be more than $\frac{3}{4}$ full, therefore increasing the chances of index collisions. To prevent that, Python would allocate a new hash table with 16 buckets, and reinsert all elements there.

All this may seem like a lot of work, but even with millions of items in a `set`, many insertions happen with no collisions, and the average number of collisions per insertion is between one and two. Under normal usage, even the unluckiest elements can be placed after a handful of collisions are resolved.

Now, given what we've seen so far, follow the flowchart in [Figure 3-4](#) to answer the following puzzle without using the computer.

Given the following `set`, what happens when you add an integer `1` to it?

```
>>> s = {1.0, 2.0, 3.0}  
>>> s.add(1)
```



How many elements are in `s` now? Does `1` replace the element `1.0`?

When you have your answer, use the Python console to verify it.

SEARCHING ELEMENTS IN A HASH TABLE

Consider the `workdays` set with the hash table shown in [Figure 3-8](#). Is '`Sat`' in it? This is the simplest execution path for the expression '`Sat`' in `workdays`:

1. Call `hash('Sat')` to get a hash code. Let's say it is 4910012646790914166
2. Derive a hash table index from the hash code, using `hash_code % table_size`. In this case, the index is 6.
3. Probe offset 6: it's empty. This means '`Sat`' is not in the set. Return `False`.

Now consider the simplest path for an element that is present in the set. To evaluate '`Thu`' in `workdays`:

1. Call `hash('Tue')`. Pretend result is 6166047609348267525.
2. Compute index: `6166047609348267525 % 8` is 5.
3. Probe offset 5:
 - a. Compare hash codes. They are equal.

- b. Compare the object values. They are equal. Return True.

Collisions are handled in the way described when adding an element. In fact, the flowchart in [Figure 3-4](#) applies to searches as well, with the exception of the terminal nodes—the rectangles with rounded corners. If an empty bucket is found, the element is not present, so Python returns `False`; otherwise, when both the hash code and the values of the sought element match an element in the hash table, the return is `True`.

PRACTICAL CONSEQUENCES OF HOW SETS WORK

The `set` and `frozenset` types are both implemented with a hash table, which has these effects:

- Set elements must be hashable objects. They must implement proper `__hash__` and `__eq__` methods as described in “[What Is Hashable?](#)”.
- Membership testing is very efficient. A set may have millions of elements, but the bucket for an element can be located directly by computing the hash code of the element and deriving an index offset, with the possible overhead of a small number of probes to find a matching element or an empty bucket.
- Sets have a significant memory overhead. The most compact internal data structure for a container would be an array of pointers¹². Compared to that, a hash table adds a hash code per entry, and at least $\frac{1}{3}$ of empty buckets to minimize collisions.
- Element ordering depends on insertion order, but not in a useful or reliable way. If two elements are involved in a collision, the bucket where each is stored depends on which element is added first.
- Adding elements to a set may change the order of other elements.

That's because, as the hash table is filled, Python may need to recreate it to keep at least $\frac{1}{3}$ of the buckets empty. When this happens, elements are reinserted and different collisions may occur.

Hash table usage in `dict`

Since 2012, the implementation of the `dict` type had two major optimizations to reduce memory usage. The first one was proposed as [PEP 412 — Key-Sharing Dictionary](#) and implemented in Python 3.3¹³. The second is called “compact dict”, and landed in Python 3.6. As a side effect, the compact `dict` space optimization preserves key insertion order. In the next sections we’ll discuss the compact `dict` and the new key-sharing scheme—in this order, for easier presentation.

HOW COMPACT DICT SAVES SPACE

NOTE

This is a high level explanation of the Python `dict` implementation. One difference is that the actual usable fraction of a `dict` hash table is $\frac{1}{3}$, and not $\frac{2}{3}$ as in sets. The actual $\frac{1}{3}$ fraction would require 16 buckets to hold the 4 items in my example `dict`, and the diagrams in this section would become too tall, so I pretend the usable fraction is $\frac{2}{3}$ in these explanations. One comment in [Objects/dictobject.c](#) explains that any fraction between $\frac{1}{3}$ and $\frac{2}{3}$ “seem to work well in practice”.

Consider a `dict` holding the abbreviated names for the weekdays from 'Mon' through 'Thu', and the number of students enrolled in swimming class on each day:

```
>>> swimmers = {'Mon': 14, 'Tue': 12, 'Wed': 14, 'Thu': 11}
```

Before the compact `dict` optimization, the hash table underlying the `swimmers` dictionary would look like Figure 3-9. As you can see, in a 64-bit Python, each bucket holds three 64-bit fields: the hash code of the key, a pointer to the key object, and a pointer to the value object. That's 24 bytes per bucket.

Old dict hash table		
	hash code	pointer to key object
0	-1	
1	-1	
2	2414279730484651250	————→ 'Tue'
3	4199492796428269555	————→ 'Mon'
4	-5145319347887138165	————→ 'Wed'
5	-1	
6	-1	
7	-1139383146578602409	————→ 'Thu'

←————— 192 bits —————→

Figure 3-9. Old hash table format for a `dict` with 4 key-value pairs. Each bucket is a struct with the hash code of the key, a pointer to the key, and a pointer to the value.

The first two fields play the same role as they do in the implementation of sets. To find a key, Python computes the hash code of the key, derives an index from the key, then probes the hash table to find a bucket with a matching hash code and a matching key object. The third field provides the main feature of a `dict`: mapping a key to an arbitrary value. The key must be a hashable object, and the hash table algorithm ensures it will be unique in the `dict`. But the value may be any object—it doesn't need to be hashable or unique.

Raymond Hettinger observed that significant savings could be made if the hash code and pointers to key and value were held in an `entries` array with no empty rows, and the actual hash table were a sparse array with much smaller buckets holding indexes into the `entries` array¹⁴. In his

original message to `python-dev`, Hettinger called the hash table `indices`. The width of the buckets in `indices` varies as the `dict` grows, starting at 8-bits per bucket—enough to index up to 128 entries, while reserving negative values for special purposes, such as -1 for empty and -2 for deleted.

As an example, the `swimmers` dictionary would then be stored as shown in Figure 3-10.

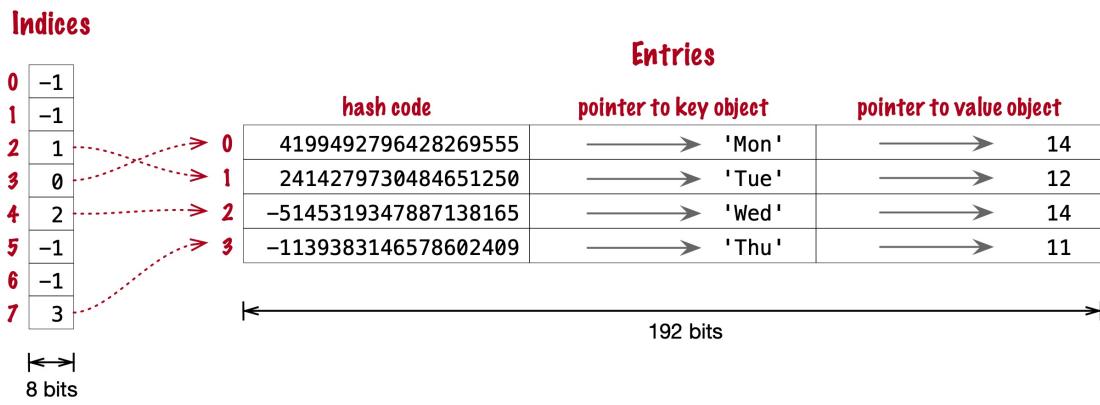


Figure 3-10. Compact storage for a `dict` with 4 key-value pairs. Hash codes and pointers to keys and values are stored in insertion order in the `entries` array, and the entry offsets derived from the hash codes are held in the `indices` sparse array, where an index value of -1 signals an empty bucket.

Assuming a 64-bit build of CPython, our 4-item `swimmers` dictionary would take 192 bytes of memory in the old scheme: 24 bytes per bucket, times 8 rows. The equivalent compact `dict` uses 104 bytes in total: 96 bytes in `entries` ($24 * 4$), plus 8 bytes for the buckets in `indices`—configured as an array of 8 bytes.

The next section describes how those two arrays are used.

ALGORITHM FOR ADDING ITEMS TO COMPACT DICT.

Step 0: set up indices

The `indices` table is initially set up as an array of signed bytes, with 8 buckets, each initialized with -1 to signal “empty bucket”. Up to 5 of these buckets will eventually hold indices to rows in the `entries` array, leaving $\frac{1}{3}$ of them with -1. The other array, `entries`, will hold key/value data with the same three fields as in the old scheme—but in insertion order.

Step 1: compute hash code for the key

To add the key-value pair ('Mon', 14) to the `swimmers` dictionary, Python first calls `hash('Mon')` to compute the hash code of that key.

Step 2: probe entries via indices

Python computes `hash('Mon') % len(indices)`. In our example, this is 3. Offset 3 in `indices` holds -1: it's an empty bucket.

Step 3: put key-value in entries, updating indices.

The `entries` array is empty, so the next available offset there is 0. Python puts 0 at offset 3 in `indices` and stores the hash code of the key, a pointer to the key object 'Mon', and a pointer to the `int` value 14 at offset 0 in `entries`. [Figure 3-11](#) shows the state of the arrays when the value of `swimmers` is `{'Mon': 14}`.

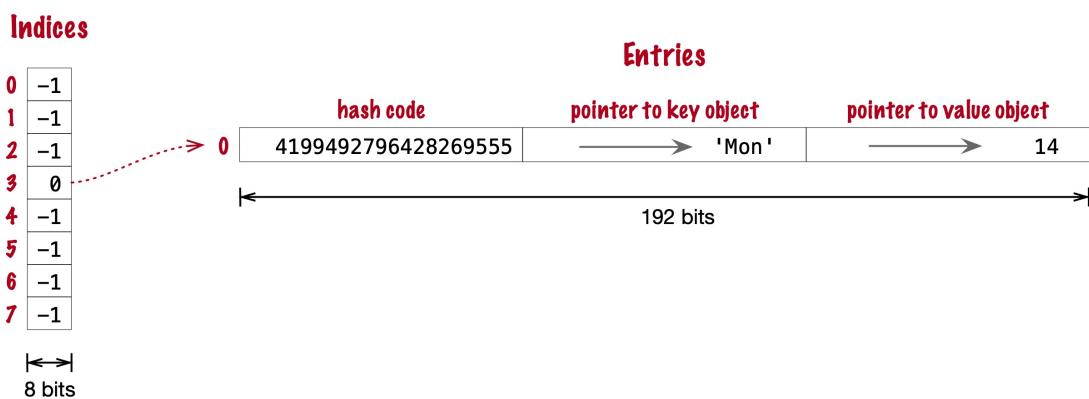


Figure 3-11. Compact storage for the `{'Mon': 14}`: `indices[3]` holds the offset of the first entry: `entries[0]`.

STEPS FOR NEXT ITEM

To add ('Tue', 12) to swimmers:

1. Compute hash code of key 'Tue'.
 2. Compute offset into `indices`, as `hash('Tue') % len(indices)`. This is 2. `indices[2]` has -1. No collision so far.
 3. Put the next available `entries` offset, 1, in `indices[2]`, then store entry at `entries[1]`.

Now the state is Figure 3-12. Note that `entries` holds the key-value pairs in insertion order.

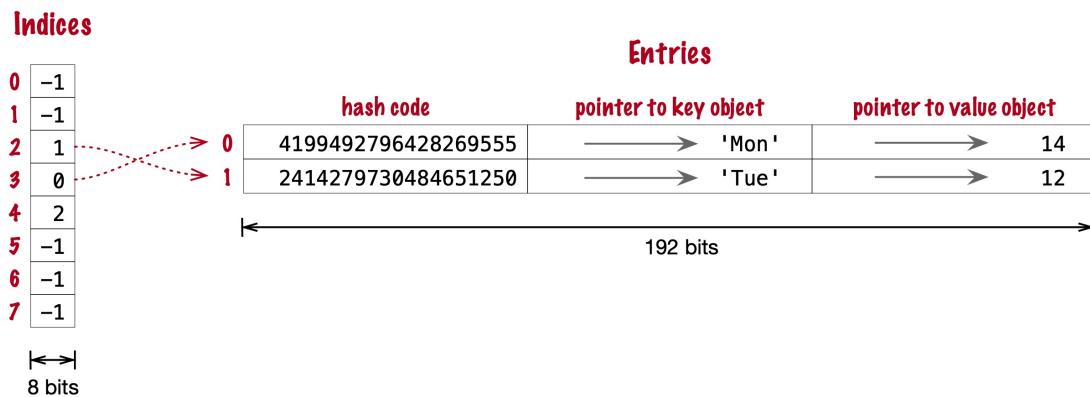


Figure 3-12. Compact storage for the `{'Mon': 14, 'Tue': 12}`.

STEPS FOR A COLLISION

1. Compute hash code of key 'Wed'.
 2. Now, `hash('Wed') % len(indices)` is 3. `indices[3]` has 0, pointing to an existing entry. Look at the hash code in `entries[0]`. That's the hash code for 'Mon', which happens to be different than the hash code for 'Wed'. This mismatch signals a collision. Probes the next index: `indices[4]`. That's

-1, so it can be used.

3. Make `indices[4] = 2`, because 2 is the next available offset at `entries`. Then fill `entries[2]` as usual.

After adding ('Wed', 14), we have Figure 3-13

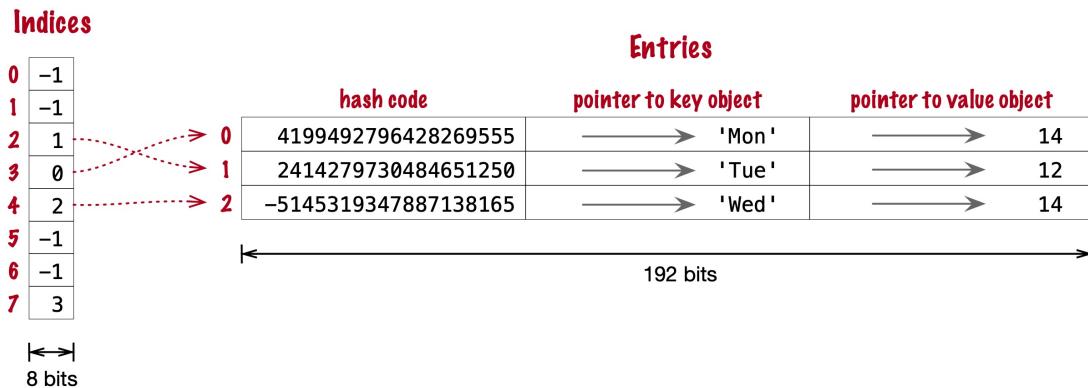


Figure 3-13. Compact storage for the {'Mon': 14, 'Tue': 12, 'Wed': 14}.

HOW A COMPACT DICT GROWS

Recall that the buckets in the `indices` array are 8 signed bytes initially, enough to hold offsets for up to 5 entries, leaving $\frac{1}{3}$ of the buckets empty. When the 6th item is added to the `dict`, `indices` is reallocated to 16 buckets—enough for 10 entry offsets. The size of `indices` is doubled as needed, while still holding signed bytes, until the time comes to add the 129th item to the `dict`. At this point, the `indices` array has 256 8-bit buckets. However, a signed byte is not enough to hold offsets after 128 entries, so the `indices` array is rebuilt to hold 256 16-bit buckets to hold signed integers—wide enough to represent offsets to 32,768 rows in the `entries` table. The next resizing happens at the 171st addition, when `indices` would become more than $\frac{2}{3}$ full. Then the number of buckets in `indices` is doubled to 512, but each bucket still 16-bits wide each. In summary, the `indices` array grows by doubling the number of buckets, and also—less often—by doubling the width of each bucket to

accomodate a growing number of rows in `entries`.

This concludes our summary of the compact `dict` implementation. I ommited many details, but now let's take a look at the other space-saving optimization for dictionaries: key-sharing.

Key-sharing dictionary

Instances of user-defined classes usually hold their attributes in a `__dict__` attribute which is a regular dictionary¹⁵. In an instance `__dict__`, the keys are the attribute names, and the values are the attribute values. Most of the time, all instances have the same attributes with different values. When that happens, 2 of the 3 fields in the `entries` table for every instance has the exact same content: the hash code of the attribute name, and a pointer to the attribute name. Only the pointer to the attribute value is different.

In [PEP 412 — Key-Sharing Dictionary](#), Mark Shannon proposed to split the storage of dictionaries used as instance `__dict__`, so that each attribute hash code and pointer is stored only once, linked to the class, and the attribute values are kept in parallel arrays of pointers attached to each instance.

Given a `Movie` class where all instances have the same attributes named `'title'`, `'release'`, `'directors'`, and `'actors'`, [Figure 3-14](#) shows the arrangement of key-sharing in a split dictionary—also implemented with the new compact layout.

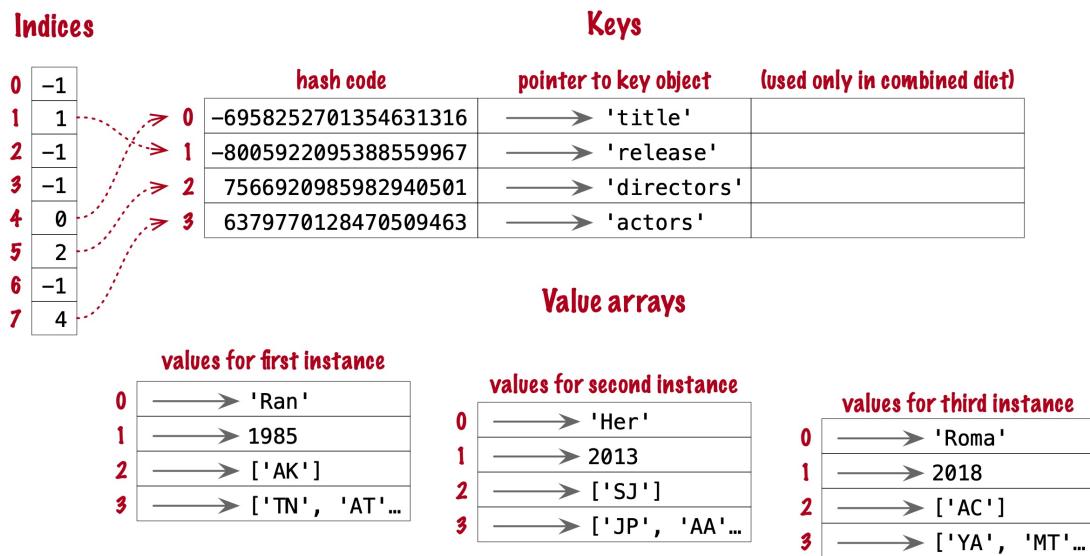


Figure 3-14. Split storage for the `__dict__` of a class and three instances.

PEP 412 introduced the terms *combined-table* to discuss the old layout and *split-table* for the proposed optimization.

The combined-table layout is still the default when you create a `dict` using literal syntax or call `dict()`. A split-table dictionary is created to fill the `__dict__` special attribute of an instance, when it is the first instance of a class. The keys table (see Figure 3-14) is then cached in the class object. This leverages the fact that most Object Oriented Python code assigns all instance attributes in the `__init__` method. That first instance (and all instances after it) will hold only its own value array. If an instance gets a new attribute not found in the shared keys table, then this instance's `__dict__` is converted to combined-table form. However, if this instance is the only one in its class, the `__dict__` is converted back to split-table, since it is assumed that further instances will have the same set of attributes and key sharing will be useful.

The `PyDictObject` struct that represents a `dict` in the CPython source code is the same for both *combined-table* and *split-table* dictionaries. When a `dict` converts from one layout to the other, the change happens

in `PyDictObject` fields, with the help of other internal data structures.

Practical Consequences of How `dict` Works

- Keys must be hashable objects. They must implement proper `__hash__` and `__eq__` methods as described in “[What Is Hashable?](#)”.
- Key searches are nearly as fast as element searches in sets.
- Item ordering is preserved in the `entries` table—this was implemented in CPython 3.6, and became an official language feature in 3.7.
- To save memory, avoid creating instance attributes outside of the `__init__` method. If all instance attributes are created in `__init__`, the `__dict__` of your instances will use the split-table layout, sharing the same indices and key entries array stored with the class.

Chapter Summary

Dictionaries are a keystone of Python. 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 it offers the `.popitem(last=False)` method option to drop and return the first item, which `dict` doesn't yet have. Also in the `collections` module is the easy-to-extend `UserDict` class.

Two powerful methods available in most mappings are `setdefault` and `update`. The `setdefault` method is used to update items holding mutable values, for example, in a `dict` of `list` values, to avoid redundant searches 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.

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 which invokes `__getitem__`.

The `collections.abc` module provides the `Mapping` and `MutableMapping` abstract base classes as standard interfaces, useful for run-time type checking. The little-known `MappingProxyType` from

the `types` module creates immutable mappings. There are also ABCs for `Set` and `MutableSet`.

Dictionary views are a great addition in Python 3, without 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 methods and operators of `frozenset`.

The hash table implementation underlying `set` is extremely fast. Understanding its logic explains why elements are apparently unordered and may even be reordered behind our backs. There is a price to pay for all this speed, and the price is in memory. Finally, we saw how optimizations in the hash tables underlying `dict` save memory and preserve key insertion order.

Further Reading

In The Python Standard Library documentation, [8.3. 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 [*Python Cookbook, Third edition*](#) (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 Gandenberger 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 the idea came 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 [Objects/dict-common.h](#), as well as the design document [Objects/dictnotes.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 (slides), 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 Donovan & Kernighan.

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 in Hans Peter Luhn and the Birth of the Hashing Algorithm.

SOAPBOX

The fundamental theorem of software engineering

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

The following quote is called—with some irony—*the fundamental theorem of software engineering*:

*Any problem in computer science can be solved with another level of indirection*¹⁶.

—David Wheeler

In the business of software, a useful variation is:

Any problem in software engineering can be solved with another level of abstraction, except for the problem of too many levels of abstraction.

—anonymous

Python's compact `dict` and the key-sharing scheme are both prime examples of indirection solving problems. But they are transparent to the end user, so they don't force us to learn new abstractions—they just make our life easier.

Simple and correct.

Syntactic sugar

Here is another famous computer science quote:

Syntactic sugar causes cancer of the semicolon.

—Alan Perlis

Some computer language purists like to dismiss syntax as unimportant, but coders know it matters in practice.

Before finding Python, I had done web programming using Perl, PHP, and JavaScript. 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 makes it easy to do configuration, table-driven implementations, and to hold data for prototyping and testing. The lack of it 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 followed the lead of Python and uses `:`. 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 `__builtin__` 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. Simple and correct.

PyCon 2017 talk; video available at <https://youtu.be/66P5FMkWoVU?t=56>

1

2 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 3-4](#).

3 This is an example of using a method as a first-class function, the subject of [Link to Come].

4

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

5

6 The exact problem with subclassing `dict` and other built-ins is covered in [Link to Come]. Dictionary views were backported to Python 2.7 and are returned by the `dict` methods `.viewkeys()`, `.viewvalues()`, and `.viewitems()`. I hope this information is of no use to you, dear reader, as Python 2.7 is now history.

7 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('set({1})')`—and a `set` call `_=dis('set([1])')`

8 The word “bucket” makes more sense to describe hash tables that hold more than one element per row. Python stores only one element per row, but we will stick with the colorful traditional term.

9 Since I just mentioned `int`, here is a CPython implementation detail: the hash code of an `int` that fits in a machine word is the value of the `int` itself, except the hash code of -1, which is -2.

10 The `hash()` built-in never returns -1 for any Python object. If `x.__hash__()` returns -1, `hash(x)` returns -2.

11 On 64-bit CPython, string hash collisions are so uncommon that I was unable to produce an example for this explanation. If you find one, let me know.

12 That’s how tuples are stored.

13 That was before I started writing the 1st edition of *Fluent Python*, but I missed it.

14 It’s ironic that the buckets in the hash table here do not contain hash codes, but only indexes to the `entries` array where the hash codes are. But, conceptually, the `index` array is really the hash table in this implementation, even if there are no hashes in its buckets.

15 Unless the class has a `__slots__` attribute, as we’ll see in chapter XXX

16 Quoted in Butler Lampson’s Turing lecture slides [Principles for Computer System Design](#)

Chapter 4. Text versus Bytes

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at fluentpython2e@ramalho.org.

Humans use text. Computers speak bytes.¹

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

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 your Python programming context, a deeper understanding of Unicode may or may not be of vital importance to you. In the end, most of the issues covered in this chapter do not affect programmers who deal only with ASCII text. But even if that is your case, 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 does 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`
- Codecs 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
- 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`
- Building emojis from character combinations

What's new in this chapter

Support for Unicode in Python 3 has been comprehensive and stable for a while, so the biggest change in this chapter is a new section on emojis—not because of changes in Python, but because of the growing popularity of emojis and emoji combinations. Unicode 13, released in 2020, supports more than 3000 emojis, and many of them that are built by combining Unicode characters. [“Multi-character emojis”](#) explains.

Also new in this 2nd edition is [“Finding characters by name”](#) including source code for utility for searching the Unicode database, a great way to find circled digits and smiling cats from the command-line.

A minor change worth mentioning is the Unicode support on Windows, which is better and simpler since Python 3.6, as we'll see in [“Encoding Defaults: A Madhouse”](#).

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

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 2020, the best definition of “character” we have is a Unicode character. Accordingly, the items you 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 you get 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 12% of the valid code points have characters assigned to them in Unicode 12.1, the standard used in Python 3.8.
- 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\xag' 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.

TIP

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 in Python 2.6.²

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)`.

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

WARNING

The fact that `my_bytes[0]` retrieves an `int` but `my_bytes[:1]` returns a `bytes` object of length 1 may be surprising, and makes it harder to support both Python 2.7 and 3 in programs that deal with binary data. But it is consistent with many other languages and also with other Python sequence types—except for `str`, which is the only sequence type where `s[0] == s[:1]`.

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

- For bytes in the printable ASCII range—from space to ~—the ASCII character itself is used.
- For bytes corresponding to tab, newline, carriage return, and \, the escape sequences `\t`, `\n`, `\r`, and `\\\` are used.
- For every other byte value, 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 a few others that depend on Unicode data, including `casifold`, `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 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 again³.

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`, `memoryview`, `array.array`); this copies the bytes from the source object to the newly created binary sequence.

WARNING

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 five short integers.

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. To read structured information in binary sequences, the `struct` module is invaluable. We'll see it working along with `bytes` and `memoryview` in [“Structs and Memory Views”](#).

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/decoder) 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\x001\x00 \x00N\x00i\x00\xf1\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
Γ	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 seven 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 `latin1` superset by Microsoft, adding useful symbols like curly quotes and the € (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 January, 2020, [W3Techs: Usage of Character Encodings for Websites] claims that 94.7% of sites use UTF-8, up from 81.4% when I wrote this paragraph in the 1st edition of *Fluent Python* 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.”

WARNING

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 Unicode 12.1, 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.

TIP

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\x00\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' `str`.
- ❸ 'cp437' can't encode the 'ã' ("a" with tilde). The default error handler—'`strict`'—raises `UnicodeEncodeError`.
- ❹ The `error='ignore'` handler silently skips characters that cannot be encoded; this is usually a very bad idea.
- ❺ When encoding, `error='replace'` substitutes unencodable characters with '?' ; data is lost, but users will get a clue that something is amiss.
- ❻ '`xmlcharrefreplace`' replaces unencodable characters with an XML entity.

NOTE

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.

TIP

Garbled characters are known as gremlins or mojibake (—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
'Monr\x91al'
>>> 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
'Monr\x91al'
```



- ❶ These bytes are the characters for “Montréal” encoded as `latin1`; '`\xe9`' is the byte for “é”.
- ❷ Decoding with '`cp1252`' (Windows 1252) works because it is a proper 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 (starting with 2.5). 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
  http://python.org/dev/peps/pep-0263/
    for details
```

Because UTF-8 is widely deployed in GNU/Linux and OSX 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!')
```

TIP

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.

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.

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? The next section answers that with a library recommendation.

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. But even then, you can never be 100% positive that a binary file is ASCII or UTF-8 just because certain bit patterns are not there.

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 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
1
```

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\x001\x00 \x00N\x00i\x00\xf1\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'\ufffe' (decimal 255, 254). Because, by design, there is no U+FFFE character in Unicode, the byte sequence b'\ufffe' 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. The character U+FEFF encoded in UTF-8 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. However, Python does not automatically assume a file is UTF-8 just because it starts with `b'\xef\xbb\xbf'`.

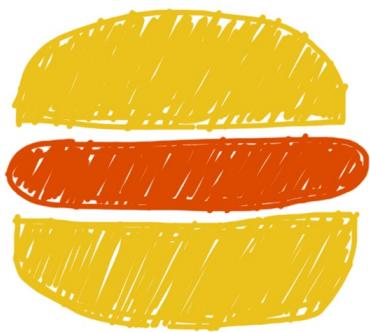
We now move on to handling text files in Python 3.

Handling Text Files

The best practice for handling text I/O is the “Unicode sandwich” ([Figure 4-2](#)).⁴ 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.

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

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.⁵

Therefore, using text files is apparently simple. But if you rely on default encodings you will get bitten.

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 Mac OSX 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.

TIP

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 ❿
<_io.BufferedReader name='cafe.txt'>
>>> fp4.read() ❾
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.
- ❻ In the Windows `cp1252` encoding, the byte 0xc3 is an “Ã” (A with tilde) and 0xa9 is the copyright sign.
- ❼ Opening the same file with the correct encoding.
- ❽ The expected result: the same four Unicode characters for 'café'.
- ❾ The '`rb`' flag opens a file for reading in binary mode.

- ➊ The returned object is a `BufferedReader` and not a `TextIOWrapper`.
- ➋ Reading that returns bytes, as expected.

TIP

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](#)”). 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.

Encoding Defaults: A Madhouse

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 sys, locale

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(expression.rjust(30), '->', repr(value))
```



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 ①
Active code page: 437
> python default_encodings.py ②
locale.getpreferredencoding() -> 'cp1252' ③
    type(my_file) -> <class '_io.TextIOWrapper'>
        my_file.encoding -> 'cp1252' ④
    sys.stdout.isatty() -> True ⑤
    sys.stdout.encoding -> 'utf-8' ⑥
        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.
- ❺ The output is going to the console, so `sys.stdout.isatty()` is `True`.
- ❻ 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 about this in the 1st edition of [_Fluent Python](#). [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 and `cmd.exe` (since Windows 1809 from October, 2018⁶). 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 file system 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`.

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.

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 = [
    '\u2026',  # HORIZONTAL ELLIPSIS (in cp1252)
    '\u221E',  # INFINITY (in cp437)
    '\u32B7',  # CIRCLED NUMBER FORTY TWO
]

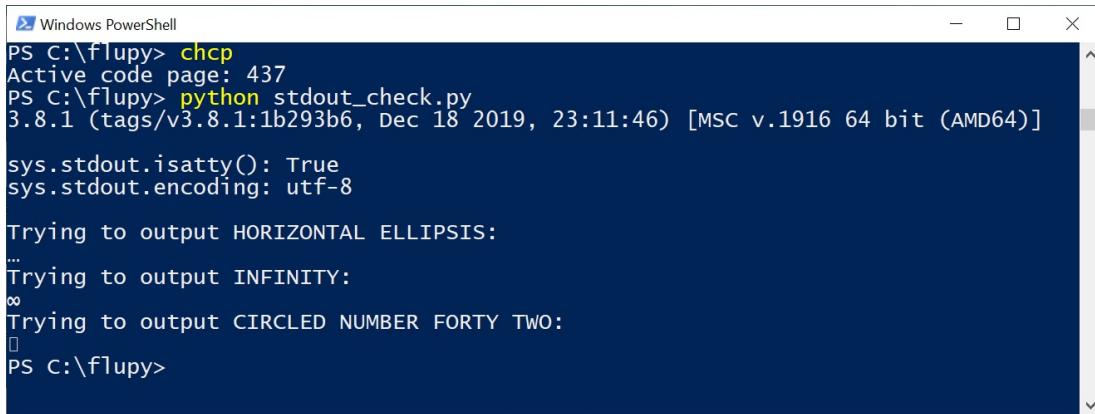
for char in test_chars:
    print(f'Trying to output {name(char)}:')
    print(char)
```

[Example 4-12](#) displays the result of `sys.stdout.isatty()`, the value of `sys.stdout.encoding`, and these three characters:

- '...' HORIZONTAL ELLIPSIS (U+2026)--exists in CP 1252 but not in CP 437

- '∞' INFINITY (U+221E)--exists in CP 437 but not in CP 1252
- '߄' CIRCLED NUMBER FORTY TWO (U+2026)--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](#).



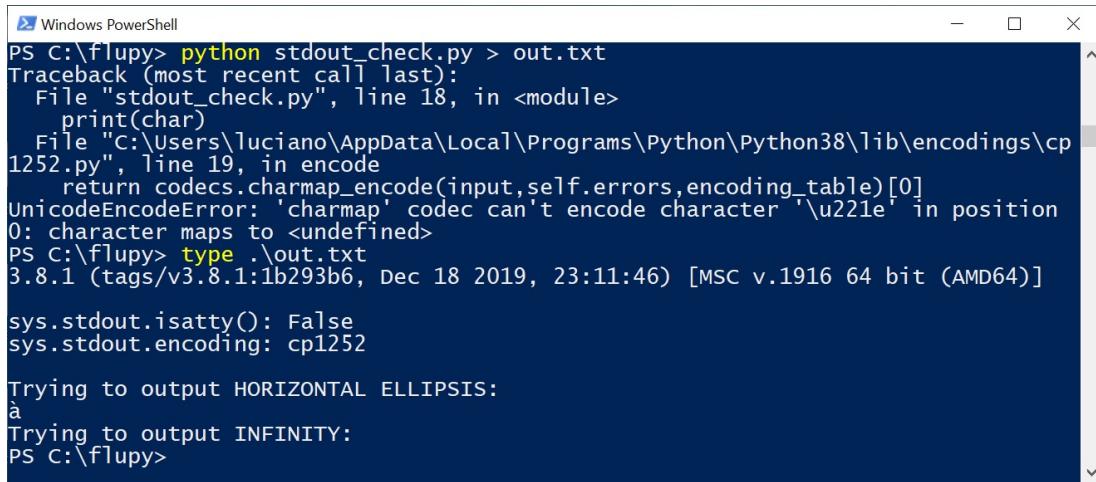
```
PS C:\flupy> chcp
Active code page: 437
PS C:\flupy> python stdout_check.py
3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)]
sys.stdout.isatty(): True
sys.stdout.encoding: utf-8

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](#).



```
Windows PowerShell
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\cp1252.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.

Then, inspecting the partially-written `out.txt`, I get two surprises:

1. 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.
2. `out.txt` was written with the UTF-16 LE encoding. This would be good, as UTF encodings support all Unicode characters—if it wasn't for the unfortunate replacement of '`...`' with '`à`'.

NOTE

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 `PYTHONLEGACYWINDOWSSSTDIO` is set to a non-empty 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`; this happens less often in Python 3, but still happens.⁷ Changing this setting is not supported.⁸
- `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. Before Python 3.6, this was MBCS on Windows, now it's UTF-8. (On this topic, a useful answer on StackOverflow is "[Difference between MBCS and UTF-8 on Windows](#)".)

NOTE

On GNU/Linux and OSX 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 Saner 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
```

The code point U+0301 is the COMBINING ACUTE ACCENT. Using it 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 to use Unicode normalization, provided by the `unicodedata.normalize` function. 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:

```
>>> from unicodedata import normalize  
>>> s1 = 'café' # composed "e" with acute accent  
>>> s2 = 'cafe\u0301' # decomposed "e" and acute accent
```

```
>>> 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
```



Western keyboards usually generate composed characters, so text typed by users will be in NFC by default. However, to be safe, it may be good to sanitize 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 uppercase omega. They are visually identical, but they compare 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
```



In the acronyms for the other two normalization forms—NFKC and

NFKD—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, 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).⁹

Here is how the NFKC works in practice:

```
>>> from unicodedata import normalize, name
>>> half = '½'
>>> normalize('NFKC', half)
'1/2'
>>> 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')
[1] 1
```

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: users may be pleased that a search for '`1/2 inch`' also finds documents containing '`½ inch`'.

WARNING

NFKC and NFKD normalization should be applied with care and only in special cases—e.g., search and indexing—and not for permanent storage, because these transformations cause data loss.

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 since Python 3.3.

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 complicated 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 = 'caf '
>>> s2 = 'cafe \u0301'
>>> s1 == s2
False
>>> nfc_equal(s1, s2)
True
>>> nfc_equal('A', 'a')
False
```

Using Normal Form C with case folding:

```
>>> s3 = 'Stra e'
>>> 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:

`http://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 friendlier, even if it is not the right spelling:

`http://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. Function to remove all combining marks (module

sanitize.py).

```
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 Etker™ caffè latte • bowl of
açaí."
>>> shave_marks(order)
'Herr Voß: • ½ cup of Etker™ caffe latte • bowl of acai.' ❶
>>> Greek = 'Ζέφυρος, Ζέφυρος'
>>> shave_marks(Greek)
'Ζεφυρος, Zefyros' ❷
```

- ❶ Only the letters “è”, “ç”, and “í” were replaced.
- ❷ Both “é” 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 sanitize.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) ①
    latin_base = False
    keepers = []
    for c in norm_txt:
        if unicodedata.combining(c) and latin_base: ②
            continue # ignore diacritic on Latin base char
        keepers.append(c) ③
        # if it isn't combining char, it's a new base char
        if not unicodedata.combining(c): ④
            latin_base = c in string.ascii_letters
    shaved = ''.join(keepers)
    return unicodedata.normalize('NFC', shaved) ⑤
```



- 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 sanitize.py from Example 4-14)

```
single_map = str.maketrans("“”‘’’‘””‘””‘””, “””‘””‘””‘””‘””, ①
                           “”””‘””*^<”””•—~””””)
multi_map = str.maketrans({ ②
                           '€': '<euro>',
                           '…': '…',
                           'Œ': 'OE',
                           '™': '(TM)',
```

```

        'œ': 'oe',
        '%': '<per mille>',
        '‡': '***',
    })

multi_map.update(single_map) ③

def dewinize(txt):
    """Replace Win1252 symbols with ASCII chars or sequences"""
    return txt.translate(multi_map) ④

def asciize(txt):
    no_marks = shave_marks_latin(dewinize(txt)) ⑤
    no_marks = no_marks.replace('ß', 'ss') ⑥
    return unicodedata.normalize('NFKC', no_marks) ⑦

```

- ① Build mapping table for char-to-char replacement.
 ② Build mapping table for char-to-string replacement.
 ③ Merge mapping tables.
 ④ `dewinize` does not affect ASCII or `latin1` text, only the Microsoft additions in to `latin1` in `cp1252`.
 ⑤ Apply `dewinize` and remove diacritical marks.
 ⑥ Replace the Eszett with “ss” (we are not using case fold here because we want to preserve the case).
 ⑦ 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 Etker™ caffè latte • bowl of açai."'
>>> dewinize(order)
'"Herr Voß: - ½ cup of Etker(TM) caffè latte - bowl of açai."' ①
>>> asciize(order)
'"Herr Voss: - 1/2 cup of Etker(TM) caffe latte - bowl of acai."' ②

```

- ❶ `dewinize` replaces curly quotes, bullets, and ™ (trademark symbol).
- ❷ `asciize` applies `dewinize`, drops diacritics, and replaces the 'ß'.

WARNING

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 `sanitize.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.

The next Unicode matter to sort out is... 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', 'açaí', 'acerola']
>>> sorted(fruits)
['acerola', 'atemoia', 'açaí', '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.¹⁰ So “cajá” is sorted as “caja,” and must come before “caju.”

The sorted `fruits` list should be:

```
['açaí', '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.”

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 sort key

```
include::code/04-text-byte/locale_sort.py
```

Running [Example 4-19](#) on GNU/Linux (Ubuntu 19.10) with the `pt_BR.UTF-8` locale installed, I get this result:

```
'pt_BR.UTF-8'  
['açaí', '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 afterwards.
- 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. They are pretty much standardized in Unix derivatives as '`language_code.encoding`', but on Windows the syntax is more complicated: `Language Name-Language Variant_Region Name.codepage`. Note that the Language Name, Language Variant, and Region Name parts can have spaces inside them, but the parts after the first are prefixed with special different characters: a hyphen, an underline character, and a dot. All parts seem to be optional except the language name. For example: `English_United States.850` means Language Name “English”, region “United States”, and code page “850”. The language and region names Windows understands are listed in the MSDN article [Language Identifier Constants and Strings](#), while [Code Page Identifiers](#) lists the numbers for the last part.¹¹
- 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 OSX, but `locale.strxfrm` never did its job.¹²

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.

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', 'açaí', 'acerola']
>>> sorted_fruits = sorted(fruits, key=coll.sort_key)
>>> sorted_fruits
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

This is friendly and just works. I tested it on GNU/Linux, OSX, and Windows. Only Python 3.X is supported at this time.

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](#).

By the way, that table is one of the many 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 `isidentifier`, `isprintable`, `isdecimal`, and `isnumeric` work. `str.casefold` also uses information from a Unicode table.

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 function¹³.

```
>>> 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](#).

```
$ ./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
(3 found)
```

Figure 4-6. Using cf.py to find smiling cats.

WARNING

Emoji support varies widely across desktop operating systems, shells, 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 support Unicode output since 2018, but as I write this in January 2020, they still don't display emojis—at least not “out of the box”.

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 test.

Example 4-21. cf.py: the character finder utility

```
#!/usr/bin/env python3
import sys
import unicodedata

FIRST, LAST = ord(' '), sys.maxunicode ①

def find(*query_words, first=FIRST, last=LAST): ②
    query = {w.upper() for w in query_words}
    count = 0
    for code in range(first, last + 1): ③
```

```

char = chr(code)                                ④
name = unicodedata.name(char, None)              ⑤
if name and query.issubset(name.split()):        ⑥
    print(f'U+{code:04X}\t{char}\t{name}')       ⑦
    count += 1
print(f'({count} found)')

def main(words):
    if words:
        find(*words)
    else:
        print('Please provide words to find.')

if __name__ == '__main__':
    main(sys.argv[1:])

```

- ① Set defaults for first and last code points to search.
- ② `find` takes zero or more `query_words`, and optional keyword-only arguments to limit the range of the search, for easier testing.
- ③ Convert the `query_words` into a set of uppercased strings.
- ④ Get Unicode character for the code.
- ⑤ Get name of character, or `None` if the code point is unassigned.
- ⑥ If there is a name, split it into a list words, then check that `query` is a subset of that.
- ⑦ 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('U+{:04x}'.format(ord(char)),           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
          format(unicodedata.numeric(char), '5.2f'), ❻
          unicodedata.name(char),                  ❼
          sep='\t')                                ❼
```



- 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 formated 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.

Code Point	Character	re_dig	isdig	isnum	Numeric Value	Name
U+0031	1	re_dig	isdig	isnum	1.00	DIGIT ONE
U+00bc	%	-	-	isnum	0.25	VULGAR FRACTION ONE QUARTER
U+00b2	²	-	isdig	isnum	2.00	SUPERSCRIPT TWO
U+0969	୩	re_dig	isdig	isnum	3.00	DEVANAGARI DIGIT THREE
U+136b	߳	-	isdig	isnum	3.00	ETHIOPIC DIGIT THREE
U+216b	XII	-	-	isnum	12.00	ROMAN NUMERAL TWELVE
U+2466	⑦	-	isdig	isnum	7.00	CIRCLED DIGIT SEVEN
U+2480	(୯)	-	-	isnum	13.00	PARENTHEΣIZED NUMBER THIRTEEN
U+3285	߶	-	-	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 in PyPI was designed to eventually replace `re` and provides better Unicode support.¹⁴ We’ll come back to the `re` module in the next section.

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 `unicodedata` module.

Next we’ll take a quick look at a new trend: 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 are 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  $1729 = 1^3 + 12^3 = 9^3 + 10^3$ .").encode('utf_8') ❹

print('Text', repr(text_str), sep='\n  ')
print('Numbers')
print('  str  :', re_numbers_str.findall(text_str))      ❺
print('  bytes:', re_numbers_bytes.findall(text_bytes))  ❻
print('Words')
print('  str  :', re_words_str.findall(text_str))        ❼
print('  bytes:', re_words_bytes.findall(text_bytes))  ⏽
```

- ❶ The first two regular expressions are of the `str` type.

- ❷ The last two are of the `bytes` type.
- ❸ Unicode text to search, containing the Tamil digits for 1729 (the logical line continues until the right parenthesis token).
- ❹ This string is joined to the previous one at compile time (see “[2.4.2. String literal concatenation](#)” in *The Python Language Reference*).
- ❺ A `bytes` string is needed to search with the `bytes` regular expressions.
- ❻ The `str` pattern `r'\d+'` matches the Tamil and ASCII digits.
- ❼ The `bytes` pattern `rb'\d+'` matches only the ASCII bytes for digits.
- ❽ The `str` pattern `r'\w+'` matches the letters, superscripts, Tamil, and ASCII digits.
- ❾ The `bytes` pattern `rb'\w+'` matches only the ASCII bytes for letters and digits.

```
$ python3 ramanujan.py
Text
'Ramanujan saw களை as 1729 = 1³ + 12³ = 9³ + 10³.'
Numbers
  str : ['களை', '1729', '1', '12', '9', '10']
  bytes: [b'1729', b'1', b'12', b'9', b'10']
Words
  str : ['Ramanujan', 'saw', 'களை', '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 of 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('.')
['abc.txt', 'digits-of-π.txt']
>>> os.listdir(b'.')
[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 file or pathnames, the `os` module provides special encoding and decoding

functions `fsencode(name_or_path)` and `os.fsdecode(name_or_path)`. Both of these functions accept an argument of type `str`, `bytes`, or—since Python 3.6—an object implementing the `os.PathLike` interface.

Enough suffering. Let’s wrap up our tour of `str` versus `bytes` with a fun topic: building emojis.

Multi-character emojis

As we saw in “[Normalizing Unicode for Saner Comparisons](#)”, it’s always been possible to produce accented characters by combining Unicode letters and diacritics. To accommodate the growing demand for emojis, this idea has been extended to produce different pictographs by combining special markers and emoji characters. Let’s start with the simplest kind of combined emoji: flags of countries.

Country flags

Throughout history, countries split, join, mutate or simply adopt new flags. The Unicode consortium found a way to avoid keeping up with those changes and outsource the problem to the systems that claim Unicode support: its character database has no country flags. Instead there is a set of 26 “regional indicator symbols letters”, from A (`U+1F1E6`) to Z (`U+1F1FF`). When you combine two of those indicator letters to form an ISO 3166-1 country code, you get the corresponding country flag—if the UI supports it. [Example 4-25](#) shows how.

Example 4-25. `two_flags.py`: combining regional indicators to produce flags

```
# REGIONAL INDICATOR SYMBOLS
```

```
RIS_A = '\U0001F1E6' # LETTER A
RIS_U = '\U0001F1FA' # LETTER U
print(RIS_A + RIS_U) # AU: Australia
print(RIS_U + RIS_A) # UA: Ukraine
print(RIS_A + RIS_A) # AA: no such country
```



Figure 4-9 shows the output of Example 4-25 on a MacOS 10.14 terminal.

\$ python3 two_flags.py



Figure 4-9. Screenshot of running `two_flags.py` from Example 4-25. The AA combination is shown as two letters A inside dashed squares.

If your program outputs a combination of indicator letters that is not recognized by the app, you get the indicators displayed as letters inside dashed squares—again, depending on the UI. See the last line in Figure 4-9.

NOTE

Europe and the United Nations are not countries, but their flags are supported by the regional indicator pairs EU and UN, respectively. England, Scotland, and Wales may or may not be separate countries by the time you read this, but they also have flags supported by Unicode. However, instead of regional indicator letters, those flags require a more complicated scheme. Read [Emoji Flags Explained](#) on Emojipedia to learn how that works.

Now let's see how emoji modifiers can be used to set the skin tone of emojis that show human faces, hands, noses etc.

Skin tones

Unicode provides a set of 5 emoji modifiers to set skin tone from pale to dark brown. They are based on the [Fitzpatrick scale](#)—developed to study the effects of ultraviolet light on human skin. [Example 4-26](#) shows the use of those modifiers to set the skin tone of the thumbs up emoji.

Example 4-26. skin.py: the thumbs up emoji by itself, followed by all available skin tone modifiers.

```
from unicodedata import name

SKIN1 = 0x1F3FB # EMOJI MODIFIER FITZPATRICK TYPE-1-2 ❶
SKINS = [chr(i) for i in range(SKIN1, SKIN1 + 5)] ❷
THUMB = '\U0001F44d' # THUMBS UP SIGN

examples = [THUMB] ❸
examples.extend(THUMB + skin for skin in SKINS) ❹

for example in examples:
    print(example, end='\t') ❺
    print(' + '.join(name(char) for char in example)) ❻
```

- ❶ EMOJI MODIFIER FITZPATRICK TYPE-1-2 is the first modifier.
- ❷ Build list with all five modifiers.
- ❸ Start list with the unmodified THUMBS UP SIGN.
- ❹ Extend list with the same emoji followed by each of the modifiers.
- ❺ Display emoji and tab.
- ❻ Display names of characters combined in the emoji, joined by ' + '.

The output of [Example 4-26](#) looks like [Figure 4-10](#) on MacOS. As you can see, the unmodified emoji has a cartoonish yellow color, while the others have more realistic skin tones.

```
$ python3 skin.py
👍 THUMBS UP SIGN
👍 THUMBS UP SIGN + EMOJI MODIFIER FITZPATRICK TYPE-1-2
👍 THUMBS UP SIGN + EMOJI MODIFIER FITZPATRICK TYPE-3
👍 THUMBS UP SIGN + EMOJI MODIFIER FITZPATRICK TYPE-4
👍 THUMBS UP SIGN + EMOJI MODIFIER FITZPATRICK TYPE-5
👍 THUMBS UP SIGN + EMOJI MODIFIER FITZPATRICK TYPE-6
```

Figure 4-10. Screenshot of Example 4-26 in the MacOS 10.14 terminal.

Let's now move to more complex emoji combinations using special markers.

Rainbow flag and other ZWJ sequences

Besides the special purpose indicators and modifiers we've seen, Unicode provides a marker that is used as glue between emojis and other characters, to produce new combinations: U+200D, ZERO WIDTH JOINER—a.k.a. ZWJ in many Unicode documents.

For example rainbow flag is built by joining the emojis WAVING WHITE FLAG and RAINBOW, as [Figure 4-11](#) shows.

```
>>> white_flag = '🏳️' # U+1F3F3 WAVING WHITE FLAG
>>> vs16 = '\uFE0F' # VARIATION SELECTOR-16
>>> zwj = '\u200D' # ZERO WIDTH JOINER
>>> rainbow = '🌈' # U+1F308 RAINBOW
>>> print(white_flag + vs16 + zwj + rainbow)
🏳️🌈
```

Figure 4-11. Making the rainbow flag in the Python console.

Unicode 13 supports more than 1100 ZWJ emoji sequences as RGI—“recommended for general interchange [...] intended to be widely supported across multiple platforms”.¹⁵ You can find the full list of RGI ZWJ emoji sequences in [emoji-zwj-sequences.txt](#) and a small sample in [Figure 4-12](#).

```

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 O
print()
E11.0 man: red hair
U+1F468 MAN + ZWG
U+1F9B0 EMOJI COMPONENT RED HAIR
E12.0 people holding hands
U+1F9D1 ADULT + ZWG
U+1F91D HANDSHAKE + ZWG
U+1F9D1 ADULT
E4.0 woman swimming: dark skin tone
U+1F3CA SWIMMER
U+1F3FF EMOJI MODIFIER FITZPATRICK TYPE-6 + ZWG
U+2640 FEMALE SIGN + V16
E4.0 woman pilot: medium-dark skin tone
U+1F469 WOMAN
U+1F3FE EMOJI MODIFIER FITZPATRICK TYPE-5 + ZWG
U+2708 AIRPLANE + V16
E2.0 family: man, woman, girl
U+1F468 MAN + ZWG
U+1F469 WOMAN + ZWG
U+1F467 GIRL
E13.0 transgender flag
U+1F3F3 WAVING WHITE FLAG + V16 + ZWG
U+26A7 MALE WITH STROKE AND MALE AND FEMALE SIGN + V16
E2.0 kiss: woman, woman
U+1F469 WOMAN + ZWG
U+2764 HEAVY BLACK HEART + V16 + ZWG
U+1F48B KISS MARK + ZWG
U+1F469 WOMAN

```

Figure 4-12. Sample ZWJ sequences generated by Example 4-27, running in a Jupyter Notebook, viewed on Firefox 72 on Ubuntu 19.10. This browser/OS combo can display all the emojis from this sample, including the newest: “people holding hands” and “transgender flag”, added in Emoji 12.0 and 13.0.

Example 4-27 is the source code that produced Figure 4-12. You can run it from your shell, but for better results I recommend pasting it inside a Jupyter Notebook to run it in a browser. Browsers often lead the way in Unicode support, and provide prettier emoji pictographs.

Example 4-27. zwj_sample.py: produce listing with a few ZWJ characters.

```

from unicodedata import name

zwj_sample = """
1F468 200D 1F9B0 |man: red hair
|E11.0
1F9D1 200D 1F91D 200D 1F9D1 |people holding hands
|E12.0
1F3CA 1F3FF 200D 2640 FEOF |woman swimming: dark skin tone
|E4.0
1F469 1F3FE 200D 2708 FEOF |woman pilot: medium-dark skin tone
|E4.0
1F468 200D 1F469 200D 1F467 |family: man, woman, girl
|E2.0
1F3F3 FEOF 200D 26A7 FEOF |transgender flag
|E13.0
1F469 200D 2764 FEOF 200D 1F48B 200D 1F469 |kiss: woman, woman
|E2.0

```

....

```
markers = {'\u200D': 'ZWJ', # ZERO WIDTH JOINER
           '\uFE0F': 'V16', # VARIATION SELECTOR-16
           }

for line in zwg_sample.strip().split('\n'):
    code, descr, version = (s.strip() for s in line.split('||'))
    chars = [chr(int(c, 16)) for c in code.split()]
    print(''.join(chars), version, descr, sep='\t', end='')

while chars:
    char = chars.pop(0)
    if char in markers:
        print(' ' + ' ' + markers[char], end=' ')
    else:
        ucode = f'U+{ord(char):04X}'
        print(f'\n\t{char}\t{ucode}\t{name(char)}', end=' ')
print()
```



One trend in modern Unicode is the addition of gender-neutral emojis such as SWIMMER (U+1F3CA) or ADULT (U+1F9D1), which can then be shown as they are, or with different gender in ZWJ sequences with the female sign ♀ (U+2640) or the male sign ♂ (U+2642). The Unicode Consortium is also moving towards more diversity in the supported family emojis. [Figure 4-13](#) is a matrix of family emojis showing current support for families with different combinations of parents and children—as of January, 2020.

A screenshot of a Firefox browser window displaying a 6x6 grid of emoji combinations. The columns and rows are labeled with emoji pairs and groups. The grid contains multiple instances of each emoji combination, such as 'boy' and 'girl' appearing in every row and column.

Figure 4-13. The table shows adult singles and couples at the top, and boys and girls on the left side. Cells have the combined emoji of a family with the parent(s) from the top and kid(s) from the left. If a combination is not supported by the browser, more than one emoji will appear inside a cell. Firefox 72 on Windows 10 is able to show all combinations.

The code I wrote to build [Figure 4-13](#) is mostly concerned with HTML formatting, but is listed in [\[Link to Come\]](#) for completeness.

Example 4-28.

Browsers follow the evolution of Unicode Emoji closely, and here no OS has a clear advantage. While preparing this chapter, I captured [Figure 4-12](#) on Ubuntu 19.10 and [Figure 4-13](#) on Windows 10, using Firefox 72 on both, because those were the OS/browser combinations with the most complete support for the emojis in those examples.

Unicode is a fascinating topic. However, now is 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.

While on the subject of source code, I presented my opinion on the debate about non-ASCII identifiers: if the maintainers of the code base want to use a human language that is not limited to ASCII characters, the identifiers should be spelled correctly. That's precisely why Python 3 accepts non-ASCII identifiers.

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:

`locale.getpreferredencoding()`,
`sys.getfilesystemencoding()`,
`sys.getdefaultencoding()`, and the encodings for the standard I/O files (e.g., `sys.stdout.encoding`). 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 OSX users, in contrast, live in a happier place where `UTF-8` is the default pretty much everywhere.

Text comparisons are surprisingly complicated because Unicode provides multiple ways of representing some characters, so normalizing is a prerequisite to 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.

Then we leveraged the Unicode database to build 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 (e.g., the `re` and `os` modules, where some functions can be called with `str` or `bytes` arguments, prompting different yet fitting results).

Finally, we saw how to produce flags, hands with different skin tones, family icons and other emoji combinations supported by Unicode.

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. Esther Nam and Travis Fischer gave an excellent PyCon 2014 talk “Character encoding and Unicode in Python: How to (╯°□°)╯ ┻━┻ with dignity” ([slides](#), [video](#)), from which I quoted this chapter’s short and sweet epigraph: “Humans use text. Computers speak bytes.” Lennart Regebro—one of this book’s technical reviewers—presents 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](#) 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, Third Edition](#) (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.

Martijn Faassen’s [“Changing the Python Default Encoding Considered Harmful”](#) and Tarek Ziadé’s [“sys.setdefaultencoding Is Evil”](#) explain why the default encoding you get from `sys.getdefaultencoding()` should never be changed, even if you discover how.

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 and Searching](#) 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](#) 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. To learn more about Unicode Emoji standards, visit the [Unicode Emoji](#) index page, which links to the [Technical Standard #51: Unicode Emoji](#) and the emoji data files, where you'll find [emoji-zwj-sequences.txt](#)—the source of the samples I used in [Figure 4-12](#).

[Emojipedia](#) is the best site to find emojis and learn about them. Besides a comprehensive searchable database, Emojipedia also has a blog including posts like [Emoji ZWJ Sequences: Three Letters, Many Possibilities](#) and [Emoji Flags Explained](#).

In 2016, the Museum of Modern Art (MoMA) in NYC 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). The culture and politics of emoji evolution in the 2010-2019 decade are the subject of Paddy Johnson’s article Emoji We Lost for Gizmodo. 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 2,693,102,686 recorded occurrences.

SOAPBOX

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 for 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”: .

Unicode Riddles

Imprecise qualifiers such as “often,” “most,” and “usually” seem to pop up whenever I write about Unicode normalization. I regret the lack of more definitive advice, but there are so many exceptions to the rules in Unicode that it is hard to be absolutely positive.

For example, the μ (micro sign) is considered a “compatibility character” but the Ω (ohm) and Å (\AA) symbols are not. The difference has practical consequences: NFC normalization—recommended for text

matching—replaces the Ω (ohm) by Ω (uppercase Grek omega) and the Å (\AA) by \AA (uppercase A with ring above). But as a “compatibility character” the μ (micro sign) is not replaced by the visually identical μ (lowercase Greek mu), except when the stronger NFKC or NFKD normalizations are applied, and these transformations are lossy.

I understand the μ (micro sign) is in Unicode because it appears in the `latin1` encoding and replacing it with the Greek mu would break round-trip conversion. After all, that’s why the micro sign is a “compatibility character.” But if the ohm and \AA symbols are not in Unicode for compatibility reasons, then why have them at all? There are already code points for the GREEK CAPITAL LETTER OMEGA and the LATIN CAPITAL LETTER A WITH RING ABOVE, which look the same and replace them on NFC normalization. Go figure.

My take after many hours studying Unicode: it is hugely complex and full of special cases, reflecting the wonderful variety of human languages and the politics of industry standards.

The power of soccer

Here is another Unicode mystery: why are England, Scotland, and Wales entitled to their own Unicode flags, but Catalonia, Pernambuco, and Texas are not? Because the first three are permanent members of the International Football Association Board which controls the rules of soccer. As such, they are allowed to enter their “national teams” in the FIFA World Cup—therefore media outlets need their flags to display tournament charts. At least that’s is my theory.

How Are str Represented in RAM?

The official Python docs avoid the issue of how the code points of a `str` are stored in memory. This is, after all, 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.

Before Python 3.3, CPython could be compiled to use either 16 or 32 bits per code point in RAM; the former was a “narrow build,” and the latter a “wide build.” To know which you have, check the value of `sys.maxunicode`: 65535 implies a “narrow build” that can’t handle code points above U+FFFF transparently. A “wide build” doesn’t have this limitation, but used a lot of more memory: 4 bytes per character, even while the vast majority of code points for Chinese ideographs fit in 2 bytes. Neither option was great, so you had to choose depending on your needs.

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, 2 or 4 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 personally why that was not such as great idea in practice: it takes a single RAT (U+1F400) to inflate an otherwise all-ASCII text into a memory-hogging internal array using 4 bytes per character. In addition, because of all the ways Unicode characters combine, the ability to retrieve an arbitrary character by

position is overrated—and extracting arbitrary slices from Unicode text is naïve at best, and often wrong. As emojis become more popular, these problems will only get worse.

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

2 Python 2.6 and 2.7 also have `bytes`, but it’s just an alias to the `str` type, and does not behave like the Python 3 `bytes` type.

3 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](#).

4 I first saw the term “Unicode sandwich” in Ned Batchelder’s excellent [“Pragmatic Unicode” talk at US PyCon 2012](#).

5 Python 2.6 or 2.7 users have to use `io.open()` to get automatic decoding/encoding when reading/writing.

6 Source: [Windows Command-Line: Unicode and UTF-8 Output Text Buffer](#).

7 While researching this subject, I did not find a list of situations when Python 3 internally converts `bytes` to `str`. Python core developer Antoine Pitrou says on the [comp.python.devel](#) list that CPython internal functions that depend on such conversions “don’t get a lot of use in py3k.”

8 The Python 2 `sys.setdefaultencoding` function was misused and is no longer documented in Python 3. It was intended for use by the core developers when the internal default encoding of Python was still undecided. In the same [comp.python.devel](#) thread, Marc-André Lemburg states that the `sys.setdefaultencoding` must never be called by user code and the only values supported by CPython are ‘`ascii`’ in Python 2 and ‘`utf-8`’ in Python 3.

9 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.

10 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.

11 Thanks to Leonardo Rochael who went beyond his duties as tech reviewer and researched these Windows details, even though he is a GNU/Linux user himself.

12 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 Mac with OSX 10.9. In summary: your mileage may vary.

13 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.

14 Although it was not better than `re` at identifying digits in this particular sample.

15 Definition quoted from [Technical Standard #51 Unicode Emoji](#).

Chapter 5. Record-like data structures

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at fluentpython2e@ramalho.org.

Data classes are like children. They are okay as a starting point, but to participate as a grownup object, they need to take some responsibility.¹

—Martin Fowler and Kent Beck

Python offers a few ways to build a simple class that is just a bunch of fields, with little or no extra functionality. That pattern is known as a “data class”—and `dataclass` is the name of a Python decorator that supports it. This chapter covers three different class builders that you may use as shortcuts to write data classes:

- `collections.namedtuple`: the simplest way—since Python 2.6;
- `typing.NamedTuple`: an alternative that allows type annotations on the fields—since Python 3.5; `class` syntax supported since 3.6;
- `@dataclasses.dataclass`: a class decorator that allows more customization than previous alternatives, adding lots of options and potential complexity—since Python 3.7.

After covering those class builders, we will discuss why *Data Class* is also the name of a code smell: a coding pattern that may be a symptom of poor object-oriented design.

The chapter ends with a section on a very different topic, but still closely related to record-like data: the `struct` module, designed to parse and build packed binary records that you may find in legacy flat-file databases, network protocols, and file headers.

NOTE

`typing.TypeDict` (since Python 3.8) may seem like another data class builder—it's described right after `typing.NamedTuple` in the [typing module documentation](#), and uses similar syntax. However, `TypedDict` does not build concrete classes that you can instantiate. It's just a way to write static annotations for variables and function arguments that are expected to accept plain dictionaries with a fixed set of keys and a specific type for the value mapped to each key.

What's new in this chapter

This chapter is new in *Fluent Python 2nd edition*. The sections “[Classic Named Tuples](#)” and “[Structs and Memory Views](#)” appeared in chapters 2 and 4 in the *1st edition*, but the rest of the chapter is completely new.

We begin with a high level overview of the three class builders.

Overview of data class builders

Consider a simple class to represent a geographic coordinate pair:

Example 5-1. class/coordinates.py

```
class Coordinate:  
    def __init__(self, lat, long):  
        self.lat = lat  
        self.long = long
```

That `Coordinate` class does the job of holding latitude and longitude attributes. Writing the `__init__` boilerplate becomes old real fast, especially if your class has more than a couple of attributes: each of them is mentioned three times! And that boilerplate doesn't buy us basic features we'd expect from a Python object:

```
>>> from coordinates import Coordinate  
>>> moscow = Coordinate(55.76, 37.62)  
>>> moscow  
<coordinates.Coordinate object at 0x107142f10> ❶  
>>> location = Coordinate(55.76, 37.62)  
>>> location == moscow ❷  
False  
>>> (location.lat, location.long) == (moscow.lat, moscow.long)  
❸  
True
```

- ❶ `__repr__` inherited from `object` is not very helpful.
- ❷ Meaningless equality; the `__eq__` method inherited from `object` compares object ids.
- ❸ Comparing two coordinates requires explicit comparison of each attribute.

The data class builders covered in this chapter provide the necessary `__init__`, `__repr__`, and `__eq__` methods automatically, as well as other useful features.

NOTE

None of the class builders discussed here depend on inheritance to do their work. Both `collections.namedtuple` and `typing.NamedTuple` build classes that are

`tuple` subclasses. `@dataclass` is a class decorator that does not affect the class hierarchy in any way. Each of them use different metaprogramming techniques to inject methods and data attributes into the class under construction.

Here is a `Coordinate` class built with `namedtuple`—a factory function that builds a subclass of `tuple` with the name and fields you specify:

```
>>> from collections import namedtuple
>>> Coordinate = namedtuple('Coordinate', 'lat long')
>>> issubclass(Coordinate, tuple)
True
>>> moscow = Coordinate(55.756, 37.617)
>>> moscow
Coordinate(lat=55.756, long=37.617) ❶
>>> moscow == Coordinate(lat=55.756, long=37.617) ❷
True
```

- ❶ Useful `__repr__`.
- ❷ Meaningful `__eq__`.

The newer `typing.NamedTuple` provides the same functionality, adding a type annotation to each field:

```
>>> import typing
>>> Coordinate = typing.NamedTuple('Coordinate', [('lat',
float), ('long', float)])
>>> issubclass(Coordinate, tuple)
True
>>> Coordinate.__annotations__
{'lat': <class 'float'>, 'long': <class 'float'>}
```

TIP

A typed named tuple can also be constructed with the fields given as keyword arguments, like this:

```
Coordinate = typing.NamedTuple('Coordinate', lat=float,  
long=float)
```

This is more readable, and also lets you provide the mapping of fields and types as `**fields_and_types`.

Since Python 3.6, `typing.NamedTuple` can also be used in a `class` statement, with type annotations written as described in [PEP 526—Syntax for Variable Annotations](#). This is much more readable, and makes it easy to override methods or add new ones. [Example 5-2](#) is the same `Coordinate` class, with a pair of `float` attributes and a custom `__str__` to display a coordinate formatted like 55.8°N, 37.6°E:

Example 5-2. `typing_namedtuple/coordinates.py`

```
from typing import NamedTuple  
  
class Coordinate(NamedTuple):  
  
    lat: float  
    long: float  
  
    def __str__(self):  
        ns = 'N' if self.lat >= 0 else 'S'  
        we = 'E' if self.long >= 0 else 'W'  
        return f'{abs(self.lat):.1f}°{ns}, {abs(self.long):.1f}°  
{we}'
```

WARNING

Although `NamedTuple` appears in the `class` statement as a superclass, it's actually not. `typing.NamedTuple` uses the advanced functionality of a metaclass² to customize the creation of the user's class. Check this out:

```
>>> issubclass(Coordinate, typing.NamedTuple)  
False
```

```
>>> issubclass(Coordinate, tuple)
True
```

In the `__init__` method generated by `typing.NamedTuple`, the fields appear as parameters in the same order they appear in the `class` statement.

Like `typing.NamedTuple`, the `dataclass` decorator supports [PEP 526](#) syntax to declare instance attributes. The decorator reads the variable annotations and automatically generates methods for your class. For comparison, check out the equivalent `Coordinate` class written with the help of the `dataclass` decorator:

Example 5-3. dataclass/coordinates.py

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Coordinate:

    lat: float
    long: float

    def __str__(self):
        ns = 'N' if self.lat >= 0 else 'S'
        we = 'E' if self.long >= 0 else 'W'
        return f'{abs(self.lat):.1f}°{ns}, {abs(self.long):.1f}°{we}'
```

Note that the body of the classes in [Example 5-2](#) and [Example 5-3](#) are identical—the difference is in the `class` statement itself. The `@dataclass` decorator does not depend on inheritance or a metaclass, so it should not interfere with your own use of these mechanisms.³ The `Coordinate` class in [Example 5-3](#) is a subclass of `object`.

Main features

The different data class builders have a lot of common. Here we'll discuss the main features they share. Table 5-1 summarizes.

Table 5-1. Selected features compared accross the three data class builders. x stands for an instance of a data class of that kind.

	<code>namedtuple</code>	<code>NamedTuple</code>	<code>dataclass</code>
mutable instances	NO	NO	YES
class statement syntax	NO	YES	YES
construct dict	<code>x._asdict()</code>	<code>x._asdict()</code>	<code>dataclasses.as_dict(x)</code>
get field names	<code>x._fields</code>	<code>x._fields</code>	<code>[f.name for f in dataclasses.fields(x)]</code>
get defaults	<code>x._field_defaults</code>	<code>x._field_defaults</code>	<code>[f.default for f in dataclasses.fields(x)]</code>
get field types	N/A	<code>x.__annotations__</code>	<code>x.__annotations__</code>
new instance with changes	<code>x._replace(...)</code>	<code>x._replace(...)</code>	<code>dataclasses.replace(x, ...)</code>
new class at runtime	<code>namedtuple(...)</code>	<code>NamedTuple(...)</code>	<code>dataclasses.make_dataclass(...)</code>

MUTABLE INSTANCES

A key difference between these class builders is that `collections.namedtuple` and `typing.NamedTuple` build `tuple` subclasses, therefore the instances are immutable. By default, `@dataclass` produces mutable classes. But the decorator accepts several keyword arguments to configure the class, including `frozen`—

shown in [Example 5-3](#). When `frozen=True`, the class will raise an exception if you try to assign values to the fields after the instance is initialized.

CLASS STATEMENT SYNTAX

`typing.NamedTuple` and `dataclass` support the regular `class` statement syntax, making it easier to add methods and docstrings to the class you are creating; `collections.namedtuple` does not support that syntax.

CONSTRUCT DICT

Both named tuple variants provide an instance method (`._as_dict`) to construct a `dict` object from the fields in a data class instance. `dataclass` avoids injecting a similar method in the data class, but provides a module-level function to do it: `dataclasses.as_dict`.

GET FIELD NAMES AND DEFAULT VALUES

All three class builders let you get the field names and default values that may be configured for them. In named tuple classes, that metadata is in the `._fields` and `._fields_defaults` class attributes. You can get the same metadata from a `dataclass` decorated class using the `fields` function from the `dataclasses` module. It returns a tuple of `Field` objects which have several attributes, including `name` and `default`.

GET FIELD TYPES

A mapping of field names to type annotations is stored in the `__annotations__` class attribute in classes defined with the help of `typing.NamedTuple` and `dataclass`.

NEW INSTANCE WITH CHANGES

Given a named tuple instance `x`, the call `x._replace(**kwargs)` returns a new instance with some attribute values replaced according to the keyword arguments given. The `dataclasses.replace(x, **kwargs)` module-level function does the same for an instance of a `dataclass` decorated class.

NEW CLASS AT RUNTIME

Although the `class` statement syntax is more readable, it is hard-coded. A framework may need to build data classes on the fly, at runtime. For that, you can use the default function call syntax of `collections.namedtuple`, which is likewise supported by `typing.NamedTuple`. The `dataclasses` module provides a `make_dataclass` function for the same purpose.

After this overview of the main features of the data class builders, let's focus on each of them in turn, starting with the simplest.

Classic Named Tuples

The `collections.namedtuple` function is a factory that builds subclasses of `tuple` enhanced with field names, a class name, and a nice `__repr__`--which helps debugging. Classes built with `namedtuple` can be used anywhere where `tuples` are needed, and in fact many functions of the Python standard library that used to return tuples now return named tuples for convenience, without affecting user's code at all.

TIP

Each instance of a class built by `namedtuple` takes exactly the same amount of

memory a tuple because the field names are stored in the class. They use less memory than a regular object because they don't store attributes as key-value pairs in one `__dict__` for each instance.

Example 5-4 shows how we could define a named tuple to hold information about a city.

Example 5-4. Defining and using a named tuple type

```
>>> from collections import namedtuple
>>> City = namedtuple('City', 'name country population
coordinates') ❶
>>> tokyo = City('Tokyo', 'JP', 36.933, (35.689722, 139.691667))
❷
>>> tokyo
City(name='Tokyo', country='JP', population=36.933, coordinates=
(35.689722,
139.691667))
>>> tokyo.population ❸
36.933
>>> tokyo.coordinates
(35.689722, 139.691667)
>>> tokyo[1]
'JP'
```

-
- ❶ Two parameters are required to create a named tuple: a class name and a list of field names, which can be given as an iterable of strings or as a single space-delimited string.
 - ❷ Field values must be passed as separate positional arguments to the constructor (in contrast, the `tuple` constructor takes a single iterable).
 - ❸ You can access the fields by name or position.

As a `tuple` subclass, `City` inherits useful methods such as `__eq__` and the special methods for comparison operators (`__gt__`, `__ge__`, etc.) which are useful for sorting sequences of `City`.

In addition to the methods inherited from `tuple`, a named tuple offers a

few attributes and methods. [Example 5-5](#) shows the most useful: the `_fields` class attribute, the class method `_make(iterable)`, and the `_asdict()` instance method.

Example 5-5. Named tuple attributes and methods (continued from the previous example)

```
>>> City._fields ❶
('name', 'country', 'population', 'location')
>>> Coordinate = namedtuple('Coordinate', 'lat long')
>>> delhi_data = ('Delhi NCR', 'IN', 21.935, Coordinate(28.613889,
77.208889))
>>> delhi = City._make(delhi_data) ❷
>>> delhi._asdict() ❸
{'name': 'Delhi NCR', 'country': 'IN', 'population': 21.935,
'location': Coordinate(lat=28.613889, long=77.208889)}
>>> import json
>>> json.dumps(delhi._asdict()) ❹
'{"name": "Delhi NCR", "country": "IN", "population": 21.935,
"location": [28.613889, 77.208889]}'
```

- ❶ `_fields` is a tuple with the field names of the class.
- ❷ `_make()` builds `City` from an iterable; `City(*delhi_data)` would do the same.
- ❸ `_asdict()` returns a `dict` built from the named tuple instance.
- ❹ `_asdict()` is useful to serialize the data in JSON format, for example.

WARNING

The `_asdict` method returned an `OrderedDict` in Python 2.7, and in Python 3.1 to 3.7. Since Python 3.8, a regular `dict` is returned—which is probably fine now that we can rely on key insertion order. If you must have an `OrderedDict`, the [_asdict documentation](#) recommends building one from the result:

```
OrderedDict(x._asdict()).
```

Since Python 3.7, `namedtuple` accepts the `defaults` keyword-only argument providing an iterable of N default values for each of the N rightmost fields of the class. [Example 5-6](#) show how to define a `Coordinate` named tuple with a default value for a `reference` field:

Example 5-6. Named tuple attributes and methods (continued from the previous example)

```
>>> Coordinate = namedtuple('Coordinate', 'lat long reference',
   defaults=['WGS84'])
>>> Coordinate(0, 0)
Coordinate(lat=0, long=0, reference='WGS84')
>>> Coordinate._field_defaults
{'reference': 'WGS84'}
```

In “[Class statement syntax](#)” I mentioned it’s easier to code methods with the class syntax supported by `typing.NamedTuple` and `@dataclass`. You can also add methods to a `namedtuple`, but it’s a hack. Skip the following box if you’re not interested in hacks.

HACKING A NAMEDTUPLE TO INJECT A METHOD

Recall how we built the `Card` class in [Example 1-1 in Chapter 1](#):

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

Later [Chapter 1](#) I wrote a `spades_high` function for sorting. It would be nice if that logic was encapsulated in method of `Card`, but adding `spades_high` to `Card` without the benefit of a class statement requires a quick hack: define the function and then assign it to a class attribute. [Example 5-7](#) shows how.

Example 5-7. `frenchdeck doctest`: Adding a class attribute and a method to `Card`, the `namedtuple` from “A Pythonic Card Deck”

```
>>> Card.suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0) ❶
>>> def spades_high(card):
...     rank_value = FrenchDeck.ranks.index(card.rank)
...     suit_value = card.suit_values[card.suit]
...     return rank_value * len(card.suit_values) + suit_value
...
>>> Card.overall_rank = spades_high ❷
>>> lowest_card = Card('2', 'clubs')
```

```
>>> highest_card = Card('A', 'spades')
>>> lowest_card.overall_rank()
0
>>> highest_card.overall_rank()
51
4
◀ ▶
1 Attach a class attribute with values for each suit.
2 spades_high will become a method; the first argument doesn't need to be named self, but it must
3 refer to the receiver (the target instance, which we usually call self).
4 Attach the function to the Cards class. It becomes a method named overall_rank.
5 It works!
6 For readability and future maintenance, its much better to be able to code methods inside a class
7 statement. But it's good to know this hack is possible, because it may come in handy.4
```

This was small detour to showcase the power of a dynamic language. Now, on to the next features of the data class builders.

Now let's check out the `typing.NamedTuple` variation.

Typed Named Tuples

The `Coordinate` class with a default field from [Example 5-6](#) can be written like this using `typing.NamedTuple`:

Example 5-8. typing_namedtuple/coordinates2.py

```
from typing import NamedTuple

class Coordinate(NamedTuple):
    lat: float
    long: float
    reference: str = 'WGS84'
```

- 1
- 2
- ◀ ▶
- 1 Every instance field must be annotated with a type.
 - 2 The `reference` instance field is annotated with a type and a default value

Classes built by `typing.NamedTuple` don't have any methods beyond those that `collections.namedtuple` also generates—and those that are inherited from `tuple`. The only difference at runtime is the presence

of the `__attributes__` class field—which Python completely ignores at runtime.

WARNING

Classes built with `typing.NamedTuple` also have a `_field_types` attribute. Since Python 3.8, that attribute is deprecated in favor of `__annotations__` which has the same information and is the canonical place to find type hints in Python objects that have them.

Given that the main feature of `typing.NamedTuple` are the type annotations, we'll take a brief look at them before resuming our exploration of data class builders.

Type hints 101

Type hints—a.k.a. type annotations—are ways to declare the expected type of function arguments, return values, and variables.

NOTE

This is a very brief introduction to type hints, just enough to make sense of the syntax and meaning of the annotations used `typing.NamedTuple` and `@dataclass` declarations. We will cover type hints for function signatures in [Link to Come] and more advanced annotations like generics, unions etc. in [Link to Come]. Here we'll mostly see hints with built-in types, such as `str`, `int`, and `float`, which are probably the most common types used to annotate fields of data classes.

The first thing you need to know about type hints is that they are not enforced at all by the Python bytecode compiler and runtime interpreter.

No runtime effect

Type annotations don't have any impact on the runtime behavior of Python programs. Check this out:

Example 5-9. Python does not enforce type hints at runtime.

```
>>> import typing
>>> class Coordinate(typing.NamedTuple):
...     lat: float
...     long: float
...
>>> trash = Coordinate('foo', None)
>>> print(trash)
Coordinate(lat='Ni!', long=None)    ⓘ
```

ⓘ I told you: no type checking at runtime!

If you type the code of Example 5-9 in a Python module, replacing the last line with `print(trash)`, it will happily run and display a meaningless `Coordinate`, with no error or warning:

```
$ python3 nocheck_demo.py
Coordinate(lat='Ni!', long=None)
```

The type hints are intended primarily to support third-party type checkers, like `Mypy` or the `PyCharm IDE` built-in type checker. These are static analysis tools: they check Python source code “at rest”, not running code.

To see the effect of type hints, you must run one of those tools on your code—like a linter. For instance, here is what `Mypy` has to say about the previous example:

```
$ mypy nocheck_demo.py
nocheck_demo.py:8: error: Argument 1 to "Coordinate" has
incompatible type "str"; expected "float"
nocheck_demo.py:8: error: Argument 2 to "Coordinate" has
```

```
incompatible type "None"; expected "float"
```



As you can see, given the definition of `Coordinate`, Mypy knows that both arguments to create an instance must be of type `float`, but the assignment to `trash` uses a `str` and `None`.⁵

Now let's talk about the syntax and meaning of type hints.

Variable annotation Syntax

Both `typing.NamedTuple` and `@dataclass` use the syntax of variable annotations defined in [PEP 526](#). This is quick introduction to that syntax in the context defining attributes in `class` statements.

The basic syntax of variable annotation is:

```
var_name: some_type
```



The type that goes after the `:` must be an identifier for one of these:

- a concrete class, for example `str` or `FrenchDeck`;
- an ABC—abstract base class;
- a type defined in the `typing` module, including special types and constructs like `Any`, `Optional`, `Union`, etc.;
- a type alias—as described in the [Type aliases](#) section of the [typing module documentation](#)

See [Acceptable type hints](#) in PEP 484 for all details.

You can also initialize the variable with a value. In a `typing.NamedTuple` or `@dataclass` declaration, that value will

become the default for that attribute, if the corresponding argument is omitted in the constructor call.

```
var_name: some_type = a_value
```



The meaning of variable annotations

We saw in “[No runtime effect](#)” that type hints have no effect at runtime. But at import time—when a module is loaded—Python does read them to build the `__annotations__` dictionary that `typing.NamedTuple` and `@dataclass` then use to enhance the class.

We’ll start this exploration with a simple class, so that we can later see what extra features are added by `typing.NamedTuple` and `@dataclass`.

Example 5-10. meaning/demo_plain.py: a plain class with type hints

```
class DemoPlainClass:
```

```
    a: int          ❶
    b: float = 1.1 ❷
    c = 'spam'      ❸
```



- ❶ `a` becomes an annotation, but is otherwise discarded.
- ❷ `b` is saved as an annotation, and also becomes a class attribute with value `1.1`.
- ❸ `c` is just a plain old class attribute, not an annotation.

We can verify that in the console, first reading the `__annotations__` of the `DemoPlainClass`, then trying to get its attributes named `a`, `b`, and `c`:

```
>>> from demo_plain import DemoPlainClass
>>> DemoPlainClass.__annotations__
```

```
{'a': <class 'int'>, 'b': <class 'float'>}  
->>> DemoPlainClass.a  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: type object 'DemoPlainClass' has no attribute  
'a'  
->>> DemoPlainClass.b  
1.1  
->>> DemoPlainClass.c  
'spam'
```

Note that the `__annotations__` special attribute is created by the interpreter to record the type hints that appear in the source code—even in a plain class.

The `a` survives only as an annotation. It doesn’t become a class attribute because no value is bound to it.⁶ The `b` and `c` are stored as class attributes because they are bound to values.

None of those three attributes will be in a new instance of `DemoPlainClass`. If you create an object `o = DemoPlainClass()`, `o.a` will raise `AttributeError`, while `o.b` and `o.c` will retrieve the class attributes with values `1.1` and `'spam'`—that’s just normal Python object behavior.

INSPECTING A TYPING.NAMEDTUPLE

Now let’s examine a class built with `typing.NamedTuple`, using the same attributes and annotations as `DemoPlainClass` from [Example 5-10](#).

Example 5-11. meaning/demo_nt.py: a class built with typing.NamedTuple.

```
import typing
```

```
class DemoNTClass(typing.NamedTuple):
```

```
    a: int      ❶
    b: float = 1.1 ❷
    c = 'spam' ❸
```

- ❶ a becomes an annotation and also an instance attribute.
- ❷ b is another annotation, and also becomes an instance attribute with default value 1.1.
- ❸ c is just a plain old class attribute; no annotation will refer to it.

Inspecting the `DemoNTClass`, we get:

```
>>> from demo_nt import DemoNTClass
>>> DemoNTClass.__annotations__
{'a': <class 'int'>, 'b': <class 'float'>}
>>> DemoNTClass.a
<_collections._tuplegetter object at 0x101f0f940>
>>> DemoNTClass.b
<_collections._tuplegetter object at 0x101f0f8b0>
>>> DemoNTClass.c
'spam'
```

Here we see the same annotations for `a` and `b` as we saw in [Example 5-10](#). But `DemoNTClass` has three class attributes `a`, `b`, and `c`. The `c` attribute is just a plain class attribute with the value '`spam`'.

The `a` and `b` class attributes are actually *descriptors*—an advanced feature covered in [\[Link to Come\]](#). For now, think of them as similar to property getters: methods that don't require the explicit call operator () to retrieve an instance attribute. In practice, this means `a` and `b` will work as read-only instance attributes—which makes sense when we recall that `DemoNTClass` instances are just a fancy tuples, and tuples are immutable.

`DemoNTClass` also gets a custom docstring:

```
>>> DemoNTClass.__doc__  
'DemoNTClass(a, b)'
```



Let's inspect an instance of `DemoNTClass`:

```
>>> nt = DemoNTClass(8)  
>>> nt.a  
8  
>>> nt.b  
1.1  
>>> nt.c  
'spam'
```



To construct `nt`, we need to give at least the `a` argument to `DemoNTClass`. The constructor also takes a `b` argument, but it has a default value of `1.1`, so it's optional. The `nt` object has the `a` and `b` attributes as expected; it doesn't have a `c` attribute, but Python retrieves it from the class, as usual.

If you try to assign values to `nt.a`, `nt.b`, `nt.c` or even `nt.z` you'll get `AttributeError`, with subtly different error messages. Try that and reflect on the messages.

INSPECTING A CLASS DECORATED WITH DATACLASS

Now we'll examine [Example 5-12](#):

Example 5-12. meaning/demo_dc.py: a class decorated with @dataclass

```
from dataclasses import dataclass  
  
@dataclass  
class DemoDataClass:  
  
    a: int
```



```
b: float = 1.1 ②  
c = 'spam' ③
```

- ① a becomes an annotation and also an instance attribute.
- ② b is another annotation, and also becomes an instance attribute with default value 1.1.
- ③ c is just a plain old class attribute; no annotation will refer to it.

Now let's check out `__annotations__`, `__doc__`, and the a, b, c attributes on `DemoDataClass`:

```
>>> from demo_dc import DemoDataClass  
>>> DemoDataClass.__annotations__  
{'a': <class 'int'>, 'b': <class 'float'>}  
>>> DemoDataClass.__doc__  
'DemoDataClass(a: int, b: float = 1.1)'  
>>> DemoDataClass.a  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: type object 'DemoDataClass' has no attribute 'a'  
>>> DemoDataClass.b  
1.1  
>>> DemoDataClass.c  
'spam'
```

The `__annotations__` and `__doc__` are not surprising. However, there is no attribute named a in `DemoDataClass`—in contrast with `DemoNTClass` from [Example 5-11](#), which has a descriptor to get a from the instances as read-only attributes (that mysterious `<_collections._tuplegetter>`). That's because the a attribute will only exist in instances of `DemoDataClass`. It will be a public attribute that we can get and set, unless the class is frozen. But b and c exist as class attributes, with b holding the default value for the b instance attribute, while c is just a class attribute that will not be bound to the instances.

Now let's see how a `DemoDataClass` instance looks like:

```
>>> dc = DemoDataClass(9)
>>> dc.a
9
>>> dc.b
1.1
>>> dc.c
'spam'
```



Again, `a` and `b` are instance attributes, and `c` is a class attribute we get via the instance.

As mentioned, `DemoDataClass` instances are mutable—and no type checking is done at runtime:

```
>>> dc.a = 10
>>> dc.b = 'oops'
```



We can do even sillier assignments:

```
>>> dc.c = 'whatever'
>>> dc.z = 'secret stash'
```



Now the `dc` instance has a `c` attribute—but that does not change the `c` class attribute. And we can add a new `z` attribute. This is normal Python behavior: regular instances can have their own attributes that don't appear in the class.⁷

More about `@dataclass`

We've only seen simple examples of `@dataclass` use so far. The decorator accepts several arguments. This is its signature:

```
@dataclass(*, init=True, repr=True, eq=True, order=False,  
          unsafe_hash=False, frozen=False)
```

The * in the first position means the remaining parameters are keyword-only. [Table 5-2](#) describes them.

Table 5-2. Keyword parameters accepted by the @dataclass decorator

option	default	meaning	notes
init	True	generate __init__ —	Ignored if __init__ is implemented by user.
repr	True	generate __repr__ —	Ignored if __repr__ is implemented by user.
eq	True	generate __eq__	Ignored if __eq__ is implemented by user.
order	False	generate __lt__, __le__, __gt__, __ge__	Raises exceptions if eq=False or any of the listed special methods are implemented by user.
unsafe_hash	False	generate __hash__ —	Complex semantics and several caveats —see: dataclass documentation .
frozen	False	make instances “immutable”	instances will be reasonably safe from accidental change, but not really immutable ^a .

^a @dataclass emulates immutability by generating __setattr__ and __delattr__ which raise `dataclass.FrozenInstanceError`—a subclass of `AttributeError`—when the user attempts to set or delete a field.

The defaults are really the most useful settings for common use cases. The options you are more likely to change from the defaults are:

- `order=True`: to allow sorting of instances of the data class;

- `frozen=True`: to protect against accidental changes to the class instances.

Given the dynamic nature of Python objects, it's not too hard for a nosy programmer to go around the protection afforded by `frozen=True`. But the necessary tricks should be easy to spot on a code review.

If `eq` and `frozen` are both true, `@dataclass` will produce a suitable `__hash__` method, so the instances will be hashable. The generated `__hash__` will use data from all fields that are not individually excluded using a field option we'll see in "[Key-sharing dictionary](#)". If `frozen=False` (the default), `@dataclass` will set `__hash__` to `None`, signalling that the instances are unhashable, therefore overriding `__hash__` from any superclass.

Regarding `unsafe_hash`, PEP 557 has this to say:

Although not recommended, you can force Data Classes to create a `__hash__` method with `unsafe_hash=True`. This might be the case if your class is logically immutable but can nonetheless be mutated. This is a specialized use case and should be considered carefully.

I will leave `unsafe_hash` at that. If you feel you must use that option, check the [`dataclasses.dataclass` documentation](#).

Further customization of the generated data class can be done at a field level.

Field options

We've already seen most basic field option: providing or not a default value with the type hint. Note that fields are read in order, and after you

declare a field with a default value, all remaining fields must also have default values. This limitation makes sense: the fields will become parameters in the generated `__init__`, and Python does not allow non-default parameters following parameters with defaults.

Mutable default values are a common source of bugs for beginning Python developers. In function definitions, a mutable default value is easily corrupted when one invocation of the function mutates the default, changing the behavior of further invocations—an issue we'll explore in “[Mutable Types as Parameter Defaults: Bad Idea](#)” ([Chapter 6](#)). Class attributes are often used as default attribute values for instances, including in data classes. And `@dataclass` uses the default values in the type hints to generate parameters with defaults for `__init__`. To prevent bugs, `@dataclass` rejects the class definition in [Example 5-13](#).

Example 5-13. `dataclass/club_wrong.py`: this class raises `ValueError`

```
@dataclass
class ClubMember:
```

```
    name: str
    guests: list = []
```



If you load the module with that `ClubMember` class, this is what you get:

```
$ python3 club_wrong.py
Traceback (most recent call last):
  File "club_wrong.py", line 4, in <module>
    class ClubMember:
      ...several lines ommitted...
ValueError: mutable default <class 'list'> for field guests is
not allowed:
use default_factory
```



The `ValueError` message explains the problem and suggests a solution: use `default_factory`. This is how to correct `ClubMember`:

Example 5-14. `dataclass/club.py`: this `ClubMember` definition works.

```
from dataclasses import dataclass, field
```

```
@dataclass
class ClubMember:

    name: str
    guests: list = field(default_factory=list)
```

In the `guests` field of [Example 5-14](#), instead of a literal `list`, the default value is set by calling the `dataclasses.field` function with `default_factory=list`. The `default_factory` parameter lets you provide a function, class, or any other callable, which will be invoked with zero arguments to build a default value each time an instance of the data class is created. This way, each instance of `ClubMember` will have its own `list`—instead of all instances sharing the same `list` from the class, which is rarely what we want and is often a bug.

WARNING

It's good that `@dataclass` rejects class definitions with a `list` default value in a field. However, be aware that it is a partial solution that only applies to `list`, `dict` and `set`. Other mutable values used as defaults will not be flagged by `@dataclass`. It's up to you to understand the problem and remember to use a default factory to set mutable default values.

If you browse the `dataclasses` module documentation, you'll see a `list` field defined with a novel syntax, as in [Example 5-15](#).

Example 5-15. dataclass/club_generic.py: this ClubMember definition is more precise

```
from dataclasses import dataclass, field
from typing import List
```

❶

```
@dataclass
class ClubMember:

    name: str
    guests: List[str] = field(default_factory=list) ❷
```

❸

❶ Import the `List` type from `typing`.

❷ `List[str]` means “a list of `str`”.

The new syntax `List[str]` is a generic type definition: the `List` class from `typing` accepts that bracket notation to specify the type of the list items. We’ll cover generics in [Link to Come]. For now, note that both Example 5-14 and Example 5-15 are correct, and the Mypy type checker does not complain about either of those class definitions. But the second one is more precise, and will allow the type checker to verify code that puts items in the list, or that read items from it.

The `default_factory` is by far the most frequently used option of the `field` function, but there are several others, listed in Table 5-3.

Table 5-3. Keyword arguments accepted by the `field` function

option	default	meaning
default	<code>_MISSING_TYPE</code>	default value for field ^a
default_factory	<code>_MISSING_TYPE</code>	0-parameter function used to produce a default
init	True	include field in parameters to <code>__init__</code>
repr	True	include field <code>__repr__</code>

hash	None	use field to compute __hash__. `footnote:[`hash=None means the field will be used in __hash__ only if compare=True.]`
compare	True	use field in comparison methods __eq__, __gt__ etc.
metadata	None	mapping with user-defined data; ignored by the @dataclass

- ^a `dataclass._MISSING_TYPE` is a sentinel value indicating the option was not provided. It exists so we can set `None` as an actual default value, a common use case.
-

The `default` option exists because the `field` call takes the place of the default value in the field annotation. If you want to create an `athlete` field with default value of `False`, and also omit that field from the `__repr__` method, you'd write this:

```
@dataclass
class ClubMember:

    name: str
    guests: list = field(default_factory=list)
    athlete: bool = field(default=False, repr=False)
```

Post-init processing

The `__init__` method generated by `@dataclass` only takes the arguments passed and assigns them—or their default values, if missing—to the instance attributes that are instance fields. But you may need to do more than that to initialize the instance. If that's the case, you can provide a `__post_init__` method. When that method exists, `@dataclass` will add code to the generated `__init__` to call `__post_init__` as the last step.

Common use cases for `__post_init__` are validation and computing field values based on other fields. We'll study a simple example that uses `__post_init__` for both of these reasons.

First, let's look at the expected behavior of a `ClubMember` subclass named `HackerClubMember`, as described by doctests in [Example 5-16](#).

Example 5-16. `dataclass/hackerclub.py`: doctests for `HackerClubMember`

```
"""
```

```
` `HackerClubMember` ` objects accept an optional ``handle`` argument::
```

```
>>> anna = HackerClubMember('Anna Ravenscroft',
handle='AnnaRaven')
>>> anna
HackerClubMember(name='Anna Ravenscroft', guests=[], handle='AnnaRaven')
```

```
If ``handle`` is omitted, it's set to the first part of the member's name::
```

```
>>> leo = HackerClubMember('Leo Rochael')
>>> leo
HackerClubMember(name='Leo Rochael', guests=[], handle='Leo')
```

```
Members must have a unique handle. The following ``leo2`` will not be created, because its ``handle`` would be 'Leo', which was taken by ``leo``::
```

```
>>> leo2 = HackerClubMember('Leo DaVinci')
Traceback (most recent call last):
...
ValueError: handle 'Leo' already exists.
```

```
To fix, ``leo2`` must be created with an explicit ``handle``::
```

```
>>> leo2 = HackerClubMember('Leo DaVinci', handle='Neo')
>>> leo2
HackerClubMember(name='Leo DaVinci', guests=[], handle='Neo')
```

```
"""
```



Note that we must provide `handle` as a keyword argument, because `HackerClubMember` inherits `name` and `guests` from `ClubMember`, and adds the `handle` field. The generated docstring for `HackerClubMember` shows the order of the fields in the constructor call:

```
>>> HackerClubMember.__doc__  
"HackerClubMember(name: str, guests: list = <factory>, handle:  
str = '')"  
[<|>]
```

Here, `<factory>` is a short way of saying that some callable will produce the default value for `guests` (in our case, the factory is the `list` class). The point is: to provide a `handle` but no `guests`, we must pass `handle` as a keyword argument.

The [Inheritance](#) section of the `dataclasses` module documentation explains how the order of the fields is computed when there are several levels of inheritance.

NOTE

In [Link to Come] we'll talk about misusing inheritance, particularly when the superclasses are not abstract. Creating a hierarchy of data classes is usually a bad idea, but it served us well here to make [Example 5-17](#) shorter, focusing on the `handle` field declaration and `__post_init__` validation.

[Example 5-17](#) is the implementation:

Example 5-17. `dataclass/hackerclub.py`: code for `HackerClubMember`.

```
from dataclasses import dataclass  
from club import ClubMember
```

```

@dataclass
class HackerClubMember(ClubMember):
    all_handles = set() ②
    handle: str = '' ③

    def __post_init__(self):
        cls = self.__class__
        if self.handle == '':
            self.handle = self.name.split()[0]
        if self.handle in cls.all_handles:
            msg = f'handle {self.handle!r} already exists.' ⑥
            raise ValueError(msg)
        cls.all_handles.add(self.handle) ⑦

```

- ① HackerClubMember extends ClubMember.
- ② all_handles is a class attribute.
- ③ handle is an instance field of type str with empty string as its default value; this makes it optional.
- ④ Get the class of the instance.
- ⑤ If self.handle is the empty string, set it to the first part of name.
- ⑥ If self.handle is in cls.all_handles, raise ValueError.
- ⑦ Add the new handle to cls.all_handles.

Example 5-17 works as intended, but is not satisfactory to a static type checker. Next, we'll see why, and how to fix it.

Typed class attributes

If we typecheck Example 5-17 with Mypy, we are reprimanded:

```
$ mypy hackerclub.py
hackerclub.py:38: error: Need type annotation for 'all_handles'
(hint: "all_handles: Set[<type>] = ...")
Found 1 error in 1 file (checked 1 source file)
```

Unfortunately, the hint provided by Mypy (version 0.750 as I write this) is not helpful in the context of `@dataclass` usage. If we add a type hint like `Set[...]` to `all_handles`, `@dataclass` will find that annotation and make `all_handles` an instance field. We saw this happening in “[Inspecting a class decorated with `dataclass`](#)”.

The work-around defined in [PEP 526—Syntax for Variable Annotations](#) is a *class variable annotation*, written with a pseudo-type named `typing.ClassVar`, which leverages the generics `[]` notation to set the type of the variable and also declare it a class attribute.

To make the type checker happy, this is how we are supposed to declare `all_handles` in [Example 5-17](#):

```
all_handles: ClassVar[Set[str]] = set()
```

That type hint is saying:

all_handles is a class attribute of type set-of-str, with an empty set as its default value.

To code that annotation, we must import `ClassVar` and `Set` from the `typing` module.

The `@dataclass` decorator doesn’t care about the types in the annotations, except in two cases, and this is one of them: if the type is `ClassVar`, an instance field will not be generated for that attribute.

The other case where the type of the field is relevant to `@dataclass` is when declaring *init-only variables*, our next topic.

Initialization variables that are not fields

Sometimes you may need to pass arguments to `__init__` that are not instance fields. Such arguments are called *init-only variables* by the [dataclasses documentation](#). To declare an argument like that, `dataclasses` module provides the pseudo-type `InitVar`, which uses the same syntax of `typing.ClassVar`. The example given in the documentation is a data class that has a field initialized from a database, and the database object must be passed to the constructor.

This is the code that illustrates the [Init-only variables](#) section:

Example 5-18. Example from the `dataclasses` module documentation.

```
@dataclass
class C:
    i: int
    j: int = None
    database: InitVar[DatabaseType] = None

    def __post_init__(self, database):
        if self.j is None and database is not None:
            self.j = database.lookup('j')

c = C(10, database=my_database)
```

Note how the `database` attribute is declared. `InitVar` will prevent `@dataclass` from treating `database` as a regular field. It will not be set as an instance attribute, and the `dataclasses.fields` function will not list it. However, `database` will be one of the arguments that the generated `__init__` will accept, and it will be also passed to `__post_init__`—if you write that method, you must add a corresponding argument to the method signature, as shown in [Example 5-18](#).

This rather long overview of `@dataclass` covered the most useful features—some of them appeared in previous sections, like “[Main features](#)” where we covered all three data class builders in parallel. The [dataclasses documentation](#) and [PEP 526 — Syntax for Variable Annotations](#) have all details.

@dataclass Example: Dublin Core Resource Record

Often, classes built with `@dataclass` will have more fields than the very short examples presented so far. Dublin Core provides the foundation for a more typical `@dataclass` example.

The Dublin Core Schema is a small set of vocabulary terms that can be used to describe digital resources (video, images, web pages, etc.), as well as physical resources such as books or CDs, and objects like artworks.

—Dublin Core on Wikipedia

The standard defines 15 optional fields, the `Resource` class in [Example 5-19](#) uses 8 of them.

Example 5-19. `dataclass/resource.py`: code for `Resource`, a class based on Dublin Core terms.

```
from dataclasses import dataclass, field
from typing import List, Optional
from enum import Enum, auto
from datetime import date
```

```
class ResourceType(Enum):
    BOOK = auto()
    EBOOK = auto()
    VIDEO = auto()
```

```
@dataclass
```

1

```

class Resource:
    """Media resource description."""
    identifier: str
    title: str = '<untitled>'
    creators: List[str] = field(default_factory=list)
    date: Optional[date] = None
    type: ResourceType = ResourceType.BOOK
    description: str = ''
    language: str = ''
    subjects: List[str] = field(default_factory=list)

```

- ❶ This `Enum` will provide type-safe values for the `Resource.type` field.
- ❷ `identifier` is the only required field.
- ❸ `title` is the first field with a default. This forces all fields below to provide defaults.
- ❹ The value of `date` can be a `datetime.date` instance, or `None`.
- ❺ The `type` field default is `ResourceType.BOOK`.

Example 5-20 is a doctest to demonstrate how a `Resource` record appears in code:

Example 5-20. `dataclass/resource.py`: code for `Resource`, a class based on Dublin Core terms.

```

>>> description = 'Improving the design of existing code'
>>> book = Resource('978-0-13-475759-9', 'Refactoring, 2nd
Edition',
...     ['Martin Fowler', 'Kent Beck'], date(2018, 11, 19),
...     ResourceType.BOOK, description, 'EN',
...     ['computer programming', 'OOP'])
>>> book # doctest: +NORMALIZE_WHITESPACE
Resource(identifier='978-0-13-475759-9', title='Refactoring,
2nd Edition',
creators=['Martin Fowler', 'Kent Beck'],
date=datetime.date(2018, 11, 19),
type=<ResourceType.BOOK: 1>, description='Improving the design
of existing code',
language='EN', subjects=['computer programming', 'OOP'])

```



The `__repr__` generated by `@dataclass` is OK, but we can do better. This is the format we want from `repr(book)`:

```
>>> book # doctest: +NORMALIZE_WHITESPACE
Resource(
    identifier = '978-0-13-475759-9',
    title = 'Refactoring, 2nd Edition',
    creators = ['Martin Fowler', 'Kent Beck'],
    date = datetime.date(2018, 11, 19),
    type = <ResourceType.BOOK: 1>,
    description = 'Improving the design of existing code',
    language = 'EN',
    subjects = ['computer programming', 'OOP'],
)
```

Example 5-21 is the code of `__repr__` to produce the format above. This example uses `doctest.fields` to get the names of the data class fields.

Example 5-21. dataclass/resource_repr.py: code for `__repr__` method implemented in the Resource class from Example 5-19.

```
def __repr__(self):
    cls = self.__class__
    cls_name = cls.__name__
    indent = ' ' * 4
    res = [f'{cls_name}('] ❶
    for f in fields(cls):
        value = getattr(self, f.name) ❷
        res.append(f'{indent}{f.name} = {value!r},') ❸
    res.append(')') ❹
    return '\n'.join(res) ❺
```

- ❶ Start the `res` list to build the output string with the class name and open parenthesis.



- ❷ For each field `f` in the class...
- ❸ Get the named attribute from the instance.
- ❹ Append an indented line with the name of the field and `repr(value)`—that's what the `!r` does.
- ❺ Append closing parenthesis.
- ❻ Build multiline string from `res` and return it.

With this example inspired by the soul of Dublin, Ohio, we conclude our tour of Python's data class builders.

Data classes are handy, but your project may suffer if you overuse them. The next section explains.

Data class as a code smell

Whether you implement a data class writing all the code yourself or leveraging one of the class builders described in this chapter, be aware that it may signal a problem in your design.

In *Refactoring, Second Edition*, Martin Fowler and Kent Beck present a catalog of “code smells”—patterns in code that may indicate the need for refactoring. The entry titled *Data Class* starts like this:

These are classes that have fields, getting and setting methods for fields, and nothing else. Such classes are dumb data holders and are often being manipulated in far too much detail by other classes.

In Fowler's personal Web site there's an illuminating post explaining what is a “code smell”. That post is very relevant to our discussion because he uses *data class* as one example of a code smell and suggests how to deal with it. Here is the post, reproduced in full.⁸

CODE SMELL

By Martin Fowler

A code smell is a surface indication that usually corresponds to a deeper problem in the system. The term was first coined by Kent Beck while helping me with my [Refactoring](#) book.

The quick definition above contains a couple of subtle points. Firstly a smell is by definition something that's quick to spot—or sniffable as I've recently put it. A long method is a good example of this—just looking at the code and my nose twitches if I see more than a dozen lines of Java.

The second is that smells don't always indicate a problem. Some long methods are just fine. You have to look deeper to see if there is an underlying problem there—smells aren't inherently bad on their own—they are often an indicator of a problem rather than the problem themselves.

The best smells are something that's easy to spot and most of time lead you to really interesting problems. Data classes (classes with all data and no behavior) are good examples of this. You look at them and ask yourself what behavior should be in this class. Then you start refactoring to move that behavior in there. Often simple questions and initial refactorings can be the vital step in turning anemic objects into something that really has class.

One of the nice things about smells is that it's easy for inexperienced people to spot them, even if they don't know enough to evaluate if there's a real problem or to correct them. I've heard of lead developers who will pick a "smell of the week" and ask people to look for the smell and bring it up with the senior members of the team. Doing it one smell at a time is a good way of gradually teaching people on the team to be better programmers.

The main idea of Object Oriented Programming is to place behavior and data together in the same code unit: a class. If a class is widely used but has no significant behavior of its own, it's possible that code dealing with its instances is scattered (and even duplicated) in methods and functions throughout the system—a recipe for maintenance headaches. That's why Fowler's refactorings to deal with a data class involve bringing responsibilities back into it.

Taking that into account, there are a couple of common scenarios where it makes sense to have a data class with little or no behavior.

Data class as scaffolding

In this scenario, the data class is an initial, simplistic implementation of a

class to jump start a new project or module. With time, the class should get its own methods, instead of relying on methods of other classes to operate on its instances. Scaffolding is temporary; eventually your custom class may become fully independent from the builder you used to start it.

Python is also used for quick problem solving and experimentation, and then it's OK leave the scaffolding in place.

Data class as intermediate representation

A data class can be useful to build records about to be exported to JSON or some other interchange format, or to hold data that was just imported, crossing some system boundary. Python's data class builders all provide a method or function to convert an instance to a plain `dict`, and you can always invoke the constructor with a `dict` used as keyword arguments expanded with `**`. Such a `dict` is very close to a JSON record.

In this scenario, the data class instances should be handled as immutable objects—even if the fields are mutable, you should not change them while they are in this intermediate form. If you do, you're losing the key benefit of having data and behavior close together. When importing/exporting requires changing values, you should implement your own builder methods instead of using the given “as dict” methods or standard constructors.

After reviewing Python's data class builders, we'll end the chapter with the `struct` module, also used for importing/exporting records, but at a much lower level.

Parsing binary records with `struct`

The `struct` module provides functions to parse fields of bytes into a tuple of Python objects, and to perform the opposite conversion, from a tuple into packed bytes. `struct` can be used with `bytes`, `bytearray`, and `memoryview` objects.

Suppose you need to read a binary file containing data about metropolitan areas, produced by a program in C with a record defined as [Example 5-22](#).

Example 5-22. MetroArea: a struct in the C language.

```
struct MetroArea {  
    int year;  
    char name[12];  
    char country[2];  
    float population;  
};
```

Here is how to read one record in that format, using `struct.unpack`:

Example 5-23. Reading a C struct in the Python console.

```
>>> from struct import unpack  
>>> FORMAT = 'i12s2sf'  
>>> data = open('metro_areas.bin', 'rb').read(24)  
>>> data  
b"\xe2\x07\x00\x00Tokyo\x00\xc5\x05\x01\x00\x00\x00JP\x00\x00\x11X"  
  
>>> unpack(FORMAT, data)  
(2018, b'Tokyo\x00\xc5\x05\x01\x00\x00\x00', b'JP', 43868228.0)
```

Note how `unpack` returns a tuple with four fields, as specified by the `FORMAT` string. The letters and numbers in `FORMAT` are [Format Characters](#) described in the `struct` module documentation.

[Table 5-4](#) explains the elements of the format string from [Example 5-23](#).

Table 5-4. Parts of the format string '`i12s2sf`'.

part	size	C type	Python type	limits to actual content
i	4 bytes	int	int	32 bits; range -2,147,483,648 to 2,147,483,647
12s	12 bytes	char[12]	bytes	length = 12
2s	2 bytes	char[2]	bytes	length = 2
f	4 bytes	float	float	32-bits; approximate range $\pm 3.4 \times 10^{38}$

One detail about the layout of `metro_areas.bin` is not clear from the code in [Example 5-22](#): size is not the only difference between the `name` and `country` fields. The `country` field always holds a 2-letter country code, but `name` is a null-terminated sequence with up to 12 bytes including the terminating b '\0'—which you can see in [Example 5-23](#) right after the word Tokyo.⁹

Now let's review a script to extract all records from `metro_areas.bin` and produce a simple report like this:

```
$ python3 metro_read.py
2018    Tokyo, JP      43,868,228
2015    Shanghai, CN   38,700,000
2015    Jakarta, ID    31,689,592
```

[Example 5-24](#) showcases the handy `struct.iter_unpack` function.

Example 5-24. metro_read.py: list all records from metro_areas.bin

```
from struct import iter_unpack

FORMAT = 'i12s2sf'

def text(field: bytes) -> str:
```

```
octets = field.split(b'\0', 1)[0]          ❸
return octets.decode('cp437')                ❹

with open('metro_areas.bin', 'rb') as fp:    ❺
    data = fp.read()

for fields in iter_unpack(FORMAT, data):      ❻
    year, name, country, pop = fields
    place = text(name) + ', ' + text(country)  ❼
    print(f'{year}\t{place}\t{pop:.0f}')
```

- ❶ The `struct` format.
- ❷ Utility function to decode and clean up the `bytes` fields; returns a `str`.¹⁰
- ❸ Handle null-terminated C string: split once on `b'\0'`, then take the first part.
- ❹ Decode `bytes` into `str`.
- ❺ Open and read the whole file in binary mode; `data` is a `bytes` object.
- ❻ `iter_unpack(...)` returns a generator that produces one tuple of fields for each sequence of bytes matching the format string.
- ❼ The `name` and `country` fields need further processing by the `text` function.

The `struct` module provides no way to specify null-terminated string fields. When processing a field like `name` in the example above, after unpacking we need to inspect the returned bytes to discard the first `b'\0'` and all bytes after it in that field. It is quite possible that bytes after the first `b'\0'` and up to the end of the field are garbage. You can actually see that in [Example 5-23](#).

Memory views can make it easier to experiment and debug programs using `struct`, as the next section explains.

Structs and Memory Views

We saw in “Memory Views” that the `memoryview` type does not let you create or store byte sequences, but provides shared memory access to slices of data from other binary sequences, packed arrays, and buffers such as Python Imaging Library (PIL) images,¹¹ without copying the bytes.

Example 5-25 shows the use of `memoryview` and `struct` together to extract the width and height of a GIF image.

Example 5-25. Using `memoryview` and `struct` to inspect a GIF image header

```
>>> import struct
>>> fmt = '<3s3sHH' ❶
>>> with open('filter.gif', 'rb') as fp:
...     img = memoryview(fp.read()) ❷
...
>>> header = img[:10] ❸
>>> bytes(header) ❹
b'GIF89a+\x02\xe6\x00'
>>> struct.unpack(fmt, header) ❺
(b'GIF', b'89a', 555, 230)
>>> del header ❻
>>> del img
```

- ❶ `struct` format: < little-endian; 3s3s two sequences of 3 bytes; HH two 16-bit integers.
- ❷ Create `memoryview` from file contents in memory...
- ❸ ...then another `memoryview` by slicing the first one; no bytes are copied here.
- ❹ Convert to `bytes` for display only; 10 bytes are copied here.
- ❺ Unpack `memoryview` into tuple of: type, version, width, and height.
- ❻ Delete references to release the memory associated with the `memoryview` instances.

Note that slicing a `memoryview` returns a new `memoryview`, without

copying bytes.¹²

I will not go deeper into `memoryview` or the `struct` module, but if you work with binary data, you'll find it worthwhile to study their docs: [Built-in Types » Memory Views](#) and [struct — Interpret bytes as packed binary data.](#)

Should we use `struct`?

I only recommend `struct` to handle data from legacy systems or standardized binary formats.

Proprietary binary records in the real world are brittle and can be corrupted easily. Our super simple example exposed one of many caveats: a string field may be limited only by its size in bytes, it may be padded by spaces, or it may contain a null-terminated string followed by random garbage up to a certain size. There is also the problem of endianness: the order of the bytes used to represent integers and floats, which depends on the CPU architecture.

If you need to exchange binary data among Python systems, the `pickle` module is the easiest way, by far. If the exchange involves programs in other languages, use JSON or a multi-platform binary serialization format like [MessagePack](#) or [Protocol Buffers](#).

Chapter Summary

The main topic of this chapter were the data class builders `collections.namedtuple`, `typing.NamedTuple` and `dataclasses.dataclass`. We saw that each of them generate data classes from descriptions provided as arguments to a factory function or from `class` statements with type hints—in the case of the latter two. In particular, both named tuple variants produce `tuple` subclasses, adding only the ability to access fields by name, and providing a `_fields` class attribute listing the field names as a tuple of strings.

Next we studied the main features of the three class builders side by side, including how to extract instance data as a `dict`, how to get the names and default values of fields, and how to make a new instance from an existing one.

This prompted our first look into type hints, particularly those used to annotate attributes in a `class` statement, using the notation introduced in Python 3.6 with [PEP 526—Syntax for Variable Annotations](#). Probably the most surprising aspect of type hints in general is the fact that they have no effect at all at runtime. Python remains a dynamic language. External tools, like Mypy, are needed to take advantage of typing information to detect errors via static analysis of the source code. After a basic overview of the syntax from PEP 526, we studied the effect of annotations in a plain class and in classes built by `typing.NamedTuple` and `@dataclass`.

Next we covered the most commonly used features provided by `@dataclass` and the `default_factory` option of the

`dataclasses.field` function. We also looked into the special pseudo-type hints `typing.ClassVar` and `dataclasses.InitVar` that are important in the context of data classes. This main topic concluded with an example based on the Dublin Core Schema, which illustrated how to use `dataclasses.fields` to iterate over the attributes of a `Resource` instance in a custom `__repr__`.

The “[Data class as a code smell](#)” came after that, warning against possible abuse of data classes defeating a basic principle of Object Oriented Programming: data and the functions that touch it should be together in the same class. Classes with no logic may be a sign of misplaced logic.

The final topic was a brief overview of the `struct` package, used to read and write records encoded as packed byte sequences. That section ended with suggestion for alternative libraries for binary data exchange, including Python’s `pickle` and language-independend alternatives.

Further Reading

Python’s standard documentation for the data class builders we covered is very good, and has quite a few small examples.

For `@dataclass` in particular, most of [PEP 557—Data Classes](#) was copied into the `dataclasses` module documentaion. But [PEP 557](#) has a few very informative sections that were not copied, including [Why not just use namedtuple?](#), [Why not just use typing.NamedTuple?](#) and the [Rationale](#) section which concludes with this Q&A:

Where is it not appropriate to use Data Classes?

API compatibility with tuples or dicts is required. Type validation

beyond that provided by PEPs 484 and 526 is required, or value validation or conversion is required.

—Eric V. Smith, PEP 557 Rationale

Over at [RealPython.com](#), Geir Arne Hjelle wrote a very complete [Ultimate Guide to Data Classes in Python 3.7](#).

At PyCon US 2018, Raymond Hettinger presented [Dataclasses: The code generator to end all code generators](#) (video).

For more features and advanced functionality, including validation, the [attrs project](#) led by Hynek Schlawack goes way beyond [dataclasses](#), promising “classes without boilerplate” and to “bring back the joy of writing classes by relieving you from the drudgery of implementing object protocols (aka dunder methods).” The influence of attrs on `@dataclass` is acknowledged by Eric V. Smith in PEP 557. This probably includes Smith’s most important architectural decision: the use of a class decorator instead of inheritance and/or a metaclass to do the job.

Glyph—founder of the Twisted project—wrote an excellent introduction to attrs in [The One Python Library Everyone Needs](#).

The attrs documentation includes a [discussion of alternatives](#).

Regarding *Data Class* as a code smell, the best source I found was Martin Fowler’s book *Refactoring, Second Edition*. This newest version is missing the quote from the epigraph of this chapter, “Data classes are like children...”, but otherwise it’s the best edition of Fowler’s most famous book, particularly for Pythonistas because the examples are in modern JavaScript, which is closer to Python than Java—the language of the first edition.

The Web site [Refactoring Guru](#) also has a description of the [Data Class](#) code smell.

For the `struct` module, again the [Python docs](#) are good and include simple examples. But before using `struct` you should seriously consider whether it's feasible to use Python's `json` or `pickle`, or multi-language binary serialization libraries such as [MessagePack](#) and [Protocol Buffers](#).

SOAPBOX

The entry for "Guido" in the Jargon file is about Guido van Rossum. It says, among other things:

Mythically, Guido's most important attribute besides Python itself is Guido's time machine, a device he is reputed to possess because of the unnerving frequency with which user requests for new features have been met with the response "I just implemented that last night..."

For the longest time, one of the missing pieces in Python's syntax has been a quick, standard way to declare instance attributes in a class. Many Object-Oriented languages have that. Here is part of a `Point` class definition in Smalltalk:

```
Object subclass: #Point
  instanceVariableNames: 'x y'
  classVariableNames: ''
  package: 'Kernel-BasicObjects'
```

The second line lists the names of the instance attributes `x` and `y`. If there were class attributes, they would be in the third line.

Python has always offered an easy way to declare class attributes, if they have an initial value. But instance attributes are much more common, and Python coders have been forced to look into the `__init__` method to find them, always afraid that there may be instance attributes created elsewhere in the class—or even created by external functions or methods of other classes.

Now we have `@dataclass`, yay!

But they bring their own problems.

First: when you use `@dataclass`, type hints are not optional. We've been promised for the last 7 years since [PEP 484—Type Hints](#) that they would always be optional. Now we have a major new language feature that requires them. If you don't like the whole static typing trend, you may want to use [`attrs`](#) instead.

Second: the [PEP 526](#) syntax for annotating instance and class attributes reverses the established convention of class statements: everything declared at the top-level of a class block was a class attribute (methods are class attributes too). With PEP 526 and `@dataclass`, any attribute declared at the top level with a type hint becomes an instance attribute. If it has a default value, then it will also be a class

attribute. But if you write `limit = 99` at the top-level of a dataclass, with no type hints, suddenly you are back in the realm where declarations at the top-level of the class belong to the class. Finally, if you want to annotate that class attribute with a type, you can't use regular types because then it will become an instance attribute. You must resort to that pseudo-type `ClassVar` annotation:

```
limit: ClassVar[int] = 99
```

Here we are talking about the exception to the exception to the exception to the rule. This seems rather unpythonic to me.

I did not take part in the discussions leading to PEP 526 or [PEP 557—Data Classes](#), but here is an alternative syntax that I'd like to see:

```
@dataclass
class HackerClubMember:

    .name: str
    .guests: list = field(default_factory=list)
    .handle: str = ''

    all_handles = set()
```

- ❶ Instance attributes must be declared with a `.` prefix. The dot is not part of the name—it's only used in this context to denote an instance attribute.
❷ Any attribute name that doesn't have a `.` prefix is a class attribute (as they always have been).
❸ The parser would have to change to accept that. I find this quite readable, and it avoids the triple-exception-to-the-rule issue.

I wish I could borrow Guido's time machine to go back to 2017 and sell this idea do the core team.

1 From *Refactoring, First Edition*, chapter 3, *Bad Smells in Code*, *Data Class* section, page 87.

2 Metaclasses are one of the subjects covered in [Link to Come]—*Class Metaprogramming*.

3 Class decorators are covered in [Link to Come]—*Class Metaprogramming*, along with metaclasses. Both are ways of customizing class behavior beyond what is possible with inheritance.

4 If you know Ruby, you know that injecting methods is a well-known but controversial technique among Rubyists. In Python, it's not as common, because it doesn't work with any built-in type—`str`, `list`, etc. I consider this limitation of Python a blessing.

5 In the context of type hints, `None` is not the `NoneType` singleton, but an alias for `NoneType` itself. This is strange when we stop to think about it, but appeals to our intuition and makes function return annotations easier to read in the common case of functions that

```
    return None.
```

Python has no concept of *undefined*, one of the silliest mistakes in the design of JavaScript.

6 Thank Guido!

However, almost always when I see this in real code it's a bad idea. I once spent hours
7 chasing a bug that was caused by attributes sneakily stashed in instances, like contraband
across module borders. Also, setting an attribute after `__init__` may have a high memory
cost, defeating the `__dict__` optimization we saw in “[Key-sharing dictionary](#)”.

8 I am fortunate to have Martin Fowler as a colleague at ThoughtWorks, so it took just 20
minutes to get his permission.

9 `\0` and `\x00` are two valid escape sequences for the null character, `chr(0)`, in a Python
`str` or `bytes` literal.

10 This is the first example using type hints in a function signature in this book. Simple type
hints like these are quite readable and almost intuitive.

11 [Pillow](#) is PIL’s most active fork.

12 Leonardo Rochael—one of the technical reviewers—pointed out that even less byte copying
would happen if I used the `mmap` module to open the image as a memory-mapped file. That
module is outside the scope of this book, but if you read and change binary files frequently,
learning more about [mmap — Memory-mapped file support](#) will be very fruitful.

Chapter 6. Object References, Mutability, and Recycling

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at fluentpython2e@ramalho.org.

‘You are sad,’ the Knight said in an anxious tone: ‘let me sing you a song to comfort you. [...] The name of the song is called “HADDOCKS’ EYES”.’

‘Oh, that’s the name of the song, is it?’ Alice said, trying to feel interested.

‘No, you don’t understand,’ the Knight said, looking a little vexed. ‘That’s what the name is CALLED. The name really IS “THE AGED AGED MAN.”’ (adapted from Chapter VIII. ‘It’s my own Invention’).

—Lewis Carroll, Through the Looking-Glass, and What
Alice Found There

Alice and the Knight set the tone of what we will see in this chapter. The theme is the distinction between objects and their names. A name is not the object; a name is a separate thing.

We start the chapter by presenting a metaphor for variables in Python: variables are labels, not boxes. If reference variables are old news to you, the analogy may still be handy if you need to explain aliasing issues to

others.

We then discuss the concepts of object identity, value, and aliasing. A surprising trait of tuples is revealed: they are immutable but their values may change. This leads to a discussion of shallow and deep copies. References and function parameters are our next theme: the problem with mutable parameter defaults and the safe handling of mutable arguments passed by clients of our functions.

The last sections of the chapter cover garbage collection, the `del` command, and how to use weak references to “remember” objects without keeping them alive.

This is a rather dry chapter, but its topics lie at the heart of many subtle bugs in real Python programs.

What's new in this chapter

The topics covered here are very fundamental and stable. There were no changes since Python 3.4 that are worth mentioning in this *2nd edition*.

I added a example of using `is` to test for a sentinel object, and a warning about misuses of the `is` operator at the end of “Choosing Between == and is”.

This chapter used to be in Part IV, but I decided to bring it up earlier because it works better as an ending to Part II—*Data Structures*—than an opening to *Object-Oriented Idioms*.

Let's start by unlearning that a variable is like a box where you store data.

Variables Are Not Boxes

In 1997, I took a summer course on Java at MIT. The professor, Lynn Andrea Stein—an award-winning computer science educator who currently teaches at Olin College of Engineering—made the point that the usual “variables as boxes” metaphor actually hinders the understanding of reference variables in OO languages. Python variables are like reference variables in Java, so it’s better to think of them as labels attached to objects.

Example 6-1 is a simple interaction that the “variables as boxes” idea cannot explain. Figure 6-1 illustrates why the box metaphor is wrong for Python, while sticky notes provide a helpful picture of how variables actually work.

*Example 6-1. Variables *a* and *b* hold references to the same list, not copies of the list*

```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> b
[1, 2, 3, 4]
```

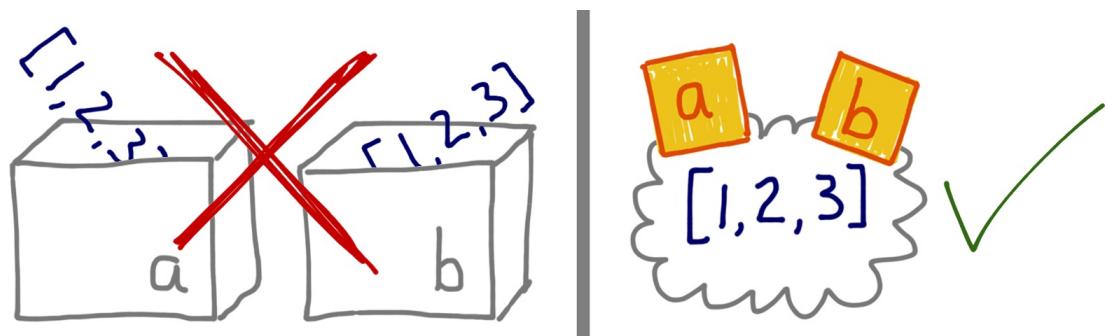


Figure 6-1. If you imagine variables are like boxes, you can't make sense of assignment in Python; instead, think of variables as sticky notes—Example 6-1 then becomes easy to explain

Prof. Stein also spoke about assignment in a very deliberate way. For example, when talking about a seesaw object in a simulation, she would

say: “Variable `s` is assigned to the seesaw,” but never “The seesaw is assigned to variable `s`.” With reference variables, it makes much more sense to say that the variable is assigned to an object, and not the other way around. After all, the object is created before the assignment.

Example 6-2 proves that the righthand side of an assignment happens first.

Example 6-2. Variables are assigned to objects only after the objects are created

```
>>> class Gizmo:  
...     def __init__(self):  
...         print('Gizmo id: %d' % id(self))  
...  
>>> x = Gizmo()  
Gizmo id: 4301489152 ❶  
>>> y = Gizmo() * 10 ❷  
Gizmo id: 4301489432 ❸  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: unsupported operand type(s) for *: 'Gizmo' and 'int'  
>>>  
>>> dir() ❹  
['Gizmo', '__builtins__', '__doc__', '__loader__', '__name__',  
 '__package__', '__spec__', 'x']
```

- ❶ The output `Gizmo id: ...` is a side effect of creating a `Gizmo` instance.
- ❷ Multiplying a `Gizmo` instance will raise an exception.
- ❸ Here is proof that a second `Gizmo` was actually instantiated before the multiplication was attempted.
- ❹ But variable `y` was never created, because the exception happened while the right-hand side of the assignment was being evaluated.

TIP

To understand an assignment in Python, always read the right-hand side first: that’s where the object is created or retrieved. After that, the variable on the left is bound to the object, like a label stuck to it. Just forget about the boxes.

Because variables are mere labels, nothing prevents an object from having several labels assigned to it. When that happens, you have *aliasing*, our next topic.

Identity, Equality, and Aliases

Lewis Carroll is the pen name of Prof. Charles Lutwidge Dodgson. Mr. Carroll is not only equal to Prof. Dodgson: they are one and the same. Example 6-3 expresses this idea in Python.

Example 6-3. charles and lewis refer to the same object

```
>>> charles = {'name': 'Charles L. Dodgson', 'born': 1832}
>>> lewis = charles ❶
>>> lewis is charles
True
>>> id(charles), id(lewis) ❷
(4300473992, 4300473992)
>>> lewis['balance'] = 950 ❸
>>> charles
{'name': 'Charles L. Dodgson', 'balance': 950, 'born': 1832} ❹
```

- ❶ `lewis` is an alias for `charles`.
- ❷ The `is` operator and the `id` function confirm it.
- ❸ Adding an item to `lewis` is the same as adding an item to `charles`.

However, suppose an impostor—let's call him Dr. Alexander Pedachenko—claims he is Charles L. Dodgson, born in 1832. His credentials may be the same, but Dr. Pedachenko is not Prof. Dodgson. Figure 6-2 illustrates this scenario.

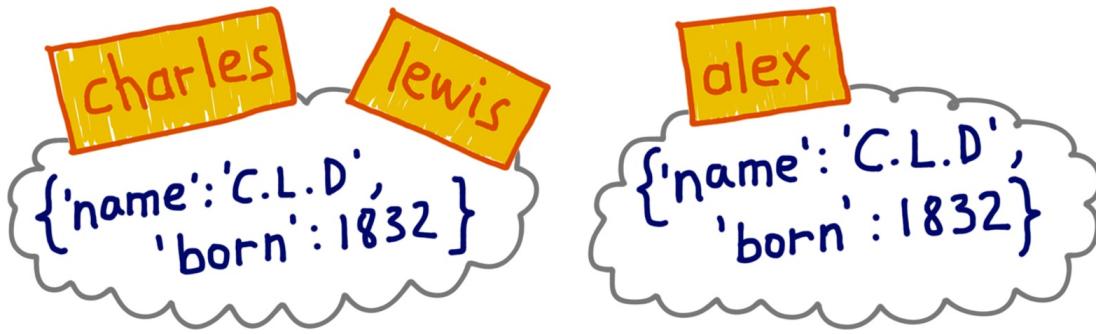


Figure 6-2. `charles` and `lewis` are bound to the same object; `alex` is bound to a separate object of equal contents

Example 6-4 implements and tests the `alex` object depicted in Figure 6-2.

Example 6-4. `alex` and `charles` compare equal, but `alex` is not `charles`

```
>>> alex = {'name': 'Charles L. Dodgson', 'born': 1832, 'balance':  
950} ❶  
>>> alex == charles ❷  
True  
>>> alex is not charles ❸  
True
```

- ❶ `alex` refers to an object that is a replica of the object assigned to `charles`.
- ❷ The objects compare equal, because of the `__eq__` implementation in the `dict` class.
- ❸ But they are distinct objects. This is the Pythonic way of writing the negative identity comparison: `a is not b`.

Example 6-3 is an example of *aliasing*. In that code, `lewis` and `charles` are aliases: two variables bound to the same object. On the other hand, `alex` is not an alias for `charles`: these variables are bound to distinct objects. The objects bound to `alex` and `charles` have the same *value*—that's what `==` compares—but they have different identities.

In *The Python Language Reference*, “3.1. Objects, values and types” states:

Every object has an identity, a type and a value. An object's identity never changes once it has been created; you may think of it as the object's address in memory. The `is` operator compares the identity of two objects; the `id()` function returns an integer representing its identity.

The real meaning of an object's ID is implementation-dependent. In CPython, `id()` returns the memory address of the object, but it may be something else in another Python interpreter. The key point is that the ID is guaranteed to be a unique numeric label, and it will never change during the life of the object.

In practice, we rarely use the `id()` function while programming. Identity checks are most often done with the `is` operator, and not by comparing IDs. Next, we'll talk about `is` versus `==`.

Choosing Between `==` and `is`

The `==` operator compares the values of objects (the data they hold), while `is` compares their identities.

We often care about values and not identities, so `==` appears more frequently than `is` in Python code.

However, if you are comparing a variable to a singleton, then it makes sense to use `is`. By far, the most common case is checking whether a variable is bound to `None`. This is the recommended way to do it:

```
x is None
```



And the proper way to write its negation is:

is not None



None by far the most common singleton we test with `is`. Sentinel objects are another example of singletons we test with `is`. Here is how you could create and test a sentinel object:

```
END_OF_DATA = object()  
# ... many lines  
def traverse(...):  
    # ... more lines  
    if node is END_OF_DATA:  
        raise StopIteration  
    # etc.
```



The `is` operator is faster than `==`, because it cannot be overloaded, so Python does not have to find and invoke special methods to evaluate it, and computing `is` is as simple as comparing two integer IDs. In contrast, `a == b` is syntactic sugar for `a.__eq__(b)`. The `__eq__` method inherited from `object` compares object IDs, so it produces the same result as `is`. But most built-in types override `__eq__` with more meaningful implementations that actually take into account the values of the object attributes. Equality may involve a lot of processing—for example, when comparing large collections or deeply nested structures.

WARNING

Usually we are more interested in object equality than identity. The **only** common proper case of `is` is checking for `None`. Most other uses I see while reviewing code are wrong. If you are not sure, use `==`. It's usually what you want, and also works with `None`—albeit not as fast.

To wrap up this discussion of identity versus equality, we'll see that the

famously immutable `tuple` is not as rigid as you may expect.

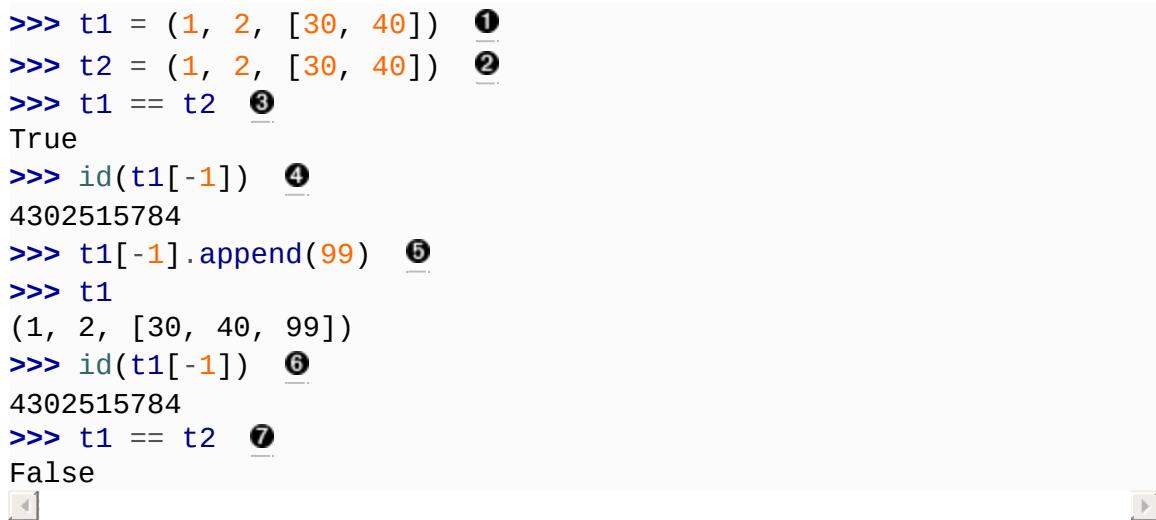
The Relative Immutability of Tuples

Tuples, like most Python collections—lists, dicts, sets, etc.—are containers: they hold references to objects.¹ If the referenced items are mutable, they may change even if the tuple itself does not. In other words, the immutability of tuples really refers to the physical contents of the `tuple` data structure (i.e., the references it holds), and does not extend to the referenced objects.

Example 6-5 illustrates the situation in which the value of a tuple changes as result of changes to a mutable object referenced in it. What can never change in a tuple is the identity of the items it contains.

Example 6-5. `t1` and `t2` initially compare equal, but changing a mutable item inside tuple `t1` makes it different

```
>>> t1 = (1, 2, [30, 40]) ❶
>>> t2 = (1, 2, [30, 40]) ❷
>>> t1 == t2 ❸
True
>>> id(t1[-1]) ❹
4302515784
>>> t1[-1].append(99) ❺
>>> t1
(1, 2, [30, 40, 99])
>>> id(t1[-1]) ❻
4302515784
>>> t1 == t2 ❼
False
```

- 
- ❶ `t1` is immutable, but `t1[-1]` is mutable.
 - ❷ Build a tuple `t2` whose items are equal to those of `t1`.
 - ❸ Although distinct objects, `t1` and `t2` compare equal, as expected.
 - ❹ Inspect the identity of the list at `t1[-1]`.
 - ❺ Modify the `t1[-1]` list in place.

- ❶ The identity of `t1[-1]` has not changed, only its value.
- ❷ `t1` and `t2` are now different.

This relative immutability of tuples is behind the riddle “[A += Assignment Puzzler](#)”. It’s also the reason why some tuples are unhashable, as we’ve seen in “[What Is Hashable?](#)”.

The distinction between equality and identity has further implications when you need to copy an object. A copy is an equal object with a different ID. But if an object contains other objects, should the copy also duplicate the inner objects, or is it OK to share them? There’s no single answer. Read on for a discussion.

Copies Are Shallow by Default

The easiest way to copy a list (or most built-in mutable collections) is to use the built-in constructor for the type itself. For example:

```
>>> l1 = [3, [55, 44], (7, 8, 9)]
>>> l2 = list(l1) ❶
>>> l2
[3, [55, 44], (7, 8, 9)]
>>> l2 == l1 ❷
True
>>> l2 is l1 ❸
False
```

- 
- 
- ❶ `list(l1)` creates a copy of `l1`.
 - ❷ The copies are equal.
 - ❸ But refer to two different objects.

For lists and other mutable sequences, the shortcut `l2 = l1[:]` also makes a copy.

However, using the constructor or `[:]` produces a *shallow copy* (i.e., the outermost container is duplicated, but the copy is filled with references to the same items held by the original container). This saves memory and causes no problems if all the items are immutable. But if there are mutable items, this may lead to unpleasant surprises.

In [Example 6-6](#), we create a shallow copy of a list containing another list and a tuple, and then make changes to see how they affect the referenced objects.

TIP

If you have a connected computer on hand, I highly recommend watching the interactive animation for [Example 6-6](#) at the [Online Python Tutor](#). As I write this, direct linking to a prepared example at pythontutor.com is not working reliably, but the tool is awesome, so taking the time to copy and paste the code is worthwhile.

Example 6-6. Making a shallow copy of a list containing another list; copy and paste this code to see it animated at the Online Python Tutor

```
l1 = [3, [66, 55, 44], (7, 8, 9)]
l2 = list(l1)
l1.append(100)
l1[1].remove(55)
print('l1:', l1)
print('l2:', l2)
l2[1] += [33, 22]
l2[2] += (10, 11)
print('l1:', l1)
print('l2:', l2)
```

- ❶ `l2` is a shallow copy of `l1`. This state is depicted in [Figure 6-3](#).
- ❷ Appending `100` to `l1` has no effect on `l2`.
- ❸ Here we remove `55` from the inner list `l1[1]`. This affects `l2`.

because `l2[1]` is bound to the same list as `l1[1]`.

- ④ For a mutable object like the list referred by `l2[1]`, the operator `+=` changes the list in place. This change is visible at `l1[1]`, which is an alias for `l2[1]`.
- ⑤ `+=` on a tuple creates a new tuple and rebinds the variable `l2[2]` here. This is the same as doing `l2[2] = l2[2] + (10, 11)`. Now the tuples in the last position of `l1` and `l2` are no longer the same object. See [Figure 6-4](#).

The output of [Example 6-6](#) is [Example 6-7](#), and the final state of the objects is depicted in [Figure 6-4](#).

Example 6-7. Output of Example 6-6

```
l1: [3, [66, 44], (7, 8, 9), 100]
l2: [3, [66, 44], (7, 8, 9)]
l1: [3, [66, 44, 33, 22], (7, 8, 9), 100]
l2: [3, [66, 44, 33, 22], (7, 8, 9, 10, 11)]
```



Frames Objects

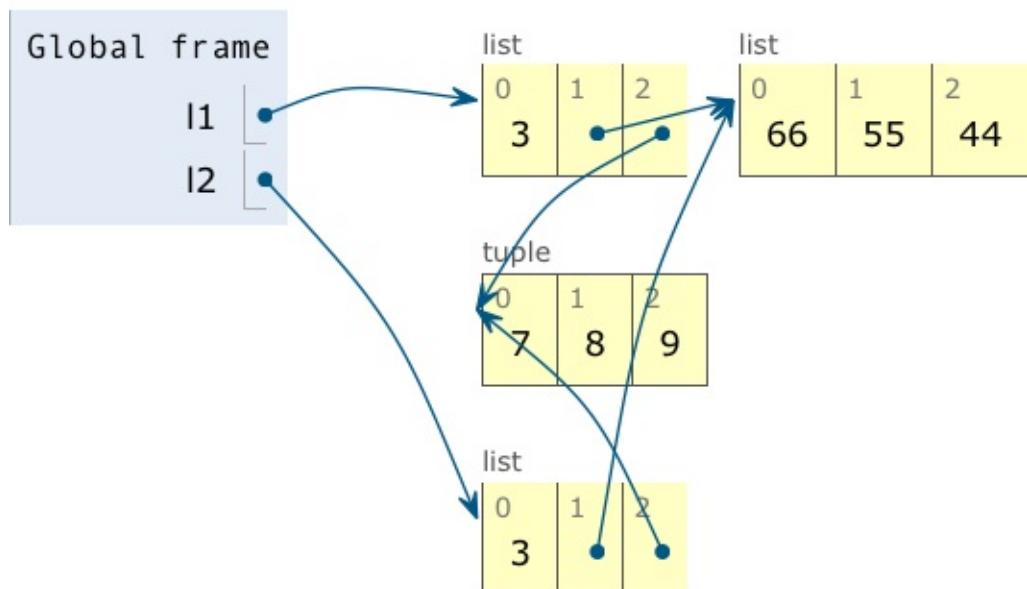


Figure 6-3. Program state immediately after the assignment `l2 = list(l1)` in Example 6-6. `l1` and `l2` refer to distinct lists, but the lists share references to the same inner list object [66, 55, 44] and tuple (7, 8, 9). (Diagram generated by the Online Python Tutor.)

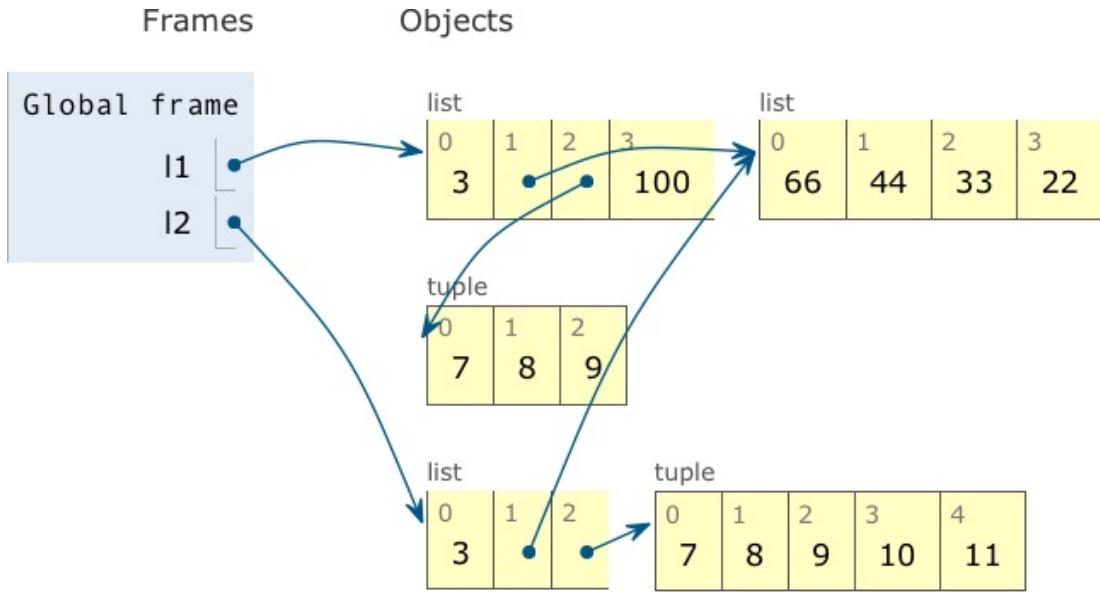


Figure 6-4. Final state of `l1` and `l2`: they still share references to the same list object, now containing [66, 44, 33, 22], but the operation `l2[2] += (10, 11)` created a new tuple with content (7, 8, 9, 10, 11), unrelated to the tuple (7, 8, 9) referenced by `l1[2]`. (Diagram generated by the Online Python Tutor.)

It should be clear now that shallow copies are easy to make, but they may or may not be what you want. How to make deep copies is our next topic.

Deep and Shallow Copies of Arbitrary Objects

Working with shallow copies is not always a problem, but sometimes you need to make deep copies (i.e., duplicates that do not share references of embedded objects). The `copy` module provides the `deepcopy` and `copy` functions that return deep and shallow copies of arbitrary objects.

To illustrate the use of `copy()` and `deepcopy()`, [Example 6-8](#) defines a simple class, `Bus`, representing a school bus that is loaded with passengers and then picks up or drops off passengers on its route.

Example 6-8. Bus picks up and drops off passengers

`class Bus:`

```
def __init__(self, passengers=None):
```

```
if passengers is None:  
    self.passengers = []  
else:  
    self.passengers = list(passengers)  
  
def pick(self, name):  
    self.passengers.append(name)  
  
def drop(self, name):  
    self.passengers.remove(name)
```

Now in the interactive [Example 6-9](#) we will create a bus object (bus1) and two clones—a shallow copy (bus2) and a deep copy (bus3)—to observe what happens as bus1 drops off a student.

Example 6-9. Effects of using copy versus deepcopy

```
>>> import copy  
>>> bus1 = Bus(['Alice', 'Bill', 'Claire', 'David'])  
>>> bus2 = copy.copy(bus1)  
>>> bus3 = copy.deepcopy(bus1)  
>>> id(bus1), id(bus2), id(bus3)  
(4301498296, 4301499416, 4301499752) ❶  
>>> bus1.drop('Bill')  
>>> bus2.passengers  
['Alice', 'Claire', 'David'] ❷  
>>> id(bus1.passengers), id(bus2.passengers), id(bus3.passengers)  
(4302658568, 4302658568, 4302657800) ❸  
>>> bus3.passengers  
['Alice', 'Bill', 'Claire', 'David'] ❹
```

- ❶ Using `copy` and `deepcopy`, we create three distinct `Bus` instances.
- ❷ After `bus1` drops 'Bill', he is also missing from `bus2`.
- ❸ Inspection of the `passengers` attributes shows that `bus1` and `bus2` share the same list object, because `bus2` is a shallow copy of `bus1`.
- ❹ `bus3` is a deep copy of `bus1`, so its `passengers` attribute refers to another list.

Note that making deep copies is not a simple matter in the general case.

Objects may have cyclic references that would cause a naïve algorithm to enter an infinite loop. The `deepcopy` function remembers the objects already copied to handle cyclic references gracefully. This is demonstrated in [Example 6-10](#).

Example 6-10. Cyclic references: b refers to a, and then is appended to a; deepcopy still manages to copy a

```
>>> a = [10, 20]
>>> b = [a, 30]
>>> a.append(b)
>>> a
[10, 20, [[...], 30]]
>>> from copy import deepcopy
>>> c = deepcopy(a)
>>> c
[10, 20, [[...], 30]]
```



Also, a deep copy may be too deep in some cases. For example, objects may refer to external resources or singletons that should not be copied. You can control the behavior of both `copy` and `deepcopy` by implementing the `__copy__()` and `__deepcopy__()` special methods as described in the [copy module documentation](#).

The sharing of objects through aliases also explains how parameter passing works in Python, and the problem of using mutable types as parameter defaults. These issues will be covered next.

Function Parameters as References

The only mode of parameter passing in Python is *call by sharing*. That is the same mode used in most OO languages, including Ruby, Smalltalk, and Java (this applies to Java reference types; primitive types use call by value). Call by sharing means that each formal parameter of the function

gets a copy of each reference in the arguments. In other words, the parameters inside the function become aliases of the actual arguments.

The result of this scheme is that a function may change any mutable object passed as a parameter, but it cannot change the identity of those objects (i.e., it cannot altogether replace an object with another). Example 6-11 shows a simple function using `+ =` on one of its parameters. As we pass numbers, lists, and tuples to the function, the actual arguments passed are affected in different ways.

Example 6-11. A function may change any mutable object it receives

```
>>> def f(a, b):
...     a += b
...     return a
...
>>> x = 1
>>> y = 2
>>> f(x, y)
3
>>> x, y ❶
(1, 2)
>>> a = [1, 2]
>>> b = [3, 4]
>>> f(a, b)
[1, 2, 3, 4]
>>> a, b ❷
([1, 2, 3, 4], [3, 4])
>>> t = (10, 20)
>>> u = (30, 40)
>>> f(t, u) ❸
(10, 20, 30, 40)
>>> t, u
((10, 20), (30, 40))
```

- 
- 
- ❶ The number `x` is unchanged.
 - ❷ The list `a` is changed.
 - ❸ The tuple `t` is unchanged.

Another issue related to function parameters is the use of mutable values for defaults, as discussed next.

Mutable Types as Parameter Defaults: Bad Idea

Optional parameters with default values are a great feature of Python function definitions, allowing our APIs to evolve while remaining backward-compatible. However, you should avoid mutable objects as default values for parameters.

To illustrate this point, in [Example 6-12](#), we take the `Bus` class from [Example 6-8](#) and change its `__init__` method to create `HauntedBus`. Here we tried to be clever and instead of having a default value of `passengers=None`, we have `passengers=[]`, thus avoiding the `if` in the previous `__init__`. This “cleverness” gets us into trouble.

Example 6-12. A simple class to illustrate the danger of a mutable default

```
class HauntedBus:  
    """A bus model haunted by ghost passengers"""  
  
    def __init__(self, passengers=[]): ❶  
        self.passengers = passengers ❷  
  
    def pick(self, name):  
        self.passengers.append(name) ❸  
  
    def drop(self, name):  
        self.passengers.remove(name)
```

- ❶ When the `passengers` argument is not passed, this parameter is bound to the default list object, which is initially empty.
- ❷ This assignment makes `self.passengers` an alias for `passengers`, which is itself an alias for the default list, when no `passengers` argument is given.
- ❸ When the methods `.remove()` and `.append()` are used with

`self.passengers` we are actually mutating the default list, which is an attribute of the function object.

Example 6-13 shows the eerie behavior of the `HauntedBus`.

Example 6-13. Buses haunted by ghost passengers

```
>>> bus1 = HauntedBus(['Alice', 'Bill'])
>>> bus1.passengers
['Alice', 'Bill']
>>> bus1.pick('Charlie')
>>> bus1.drop('Alice')
>>> bus1.passengers ❶
['Bill', 'Charlie']
>>> bus2 = HauntedBus() ❷
>>> bus2.pick('Carrie')
>>> bus2.passengers
['Carrie']
>>> bus3 = HauntedBus() ❸
>>> bus3.passengers ❹
['Carrie']
>>> bus3.pick('Dave')
>>> bus2.passengers ❺
['Carrie', 'Dave']
>>> bus2.passengers is bus3.passengers ❻
True
>>> bus1.passengers ❼
['Bill', 'Charlie']
```

- ❶ So far, so good: no surprises with `bus1`.
- ❷ `bus2` starts empty, so the default empty list is assigned to `self.passengers`.
- ❸ `bus3` also starts empty, again the default list is assigned.
- ❹ The default is no longer empty!
- ❺ Now `Dave`, picked by `bus3`, appears in `bus2`.
- ❻ The problem: `bus2.passengers` and `bus3.passengers` refer to the same list.
- ❼ But `bus1.passengers` is a distinct list.

The problem is that `HauntedBus` instances that don't get an initial

passenger list end up sharing the same passenger list among themselves.

Such bugs may be subtle. As [Example 6-13](#) demonstrates, when a `HauntedBus` is instantiated with passengers, it works as expected. Strange things happen only when a `HauntedBus` starts empty, because then `self.passengers` becomes an alias for the default value of the `passengers` parameter. The problem is that each default value is evaluated when the function is defined—i.e., usually when the module is loaded—and the default values become attributes of the function object. So if a default value is a mutable object, and you change it, the change will affect every future call of the function.

After running the lines in [Example 6-13](#), you can inspect the `HauntedBus.__init__` object and see the ghost students haunting its `__defaults__` attribute:

```
>>> dir(HauntedBus.__init__) # doctest: +ELLIPSIS
['__annotations__', '__call__', ..., '__defaults__', ...]
>>> HauntedBus.__init__.__defaults__
(['Carrie', 'Dave'])
```

Finally, we can verify that `bus2.passengers` is an alias bound to the first element of the `HauntedBus.__init__.__defaults__` attribute:

```
>>> HauntedBus.__init__.__defaults__[0] is bus2.passengers
True
```

The issue with mutable defaults explains why `None` is often used as the default value for parameters that may receive mutable values. In [Example 6-8](#), `__init__` checks whether the `passengers` argument is

`None`, and assigns a new empty list to `self.passengers`. As explained in the following section, if `passengers` is not `None`, the correct implementation assigns a copy of it to `self.passengers`. Let's now take a closer look.

Defensive Programming with Mutable Parameters

When you are coding a function that receives a mutable parameter, you should carefully consider whether the caller expects the argument passed to be changed.

For example, if your function receives a `dict` and needs to modify it while processing it, should this side effect be visible outside of the function or not? Actually it depends on the context. It's really a matter of aligning the expectation of the coder of the function and that of the caller.

The last bus example in this chapter shows how a `TwilightBus` breaks expectations by sharing its passenger list with its clients. Before studying the implementation, see in [Example 6-14](#) how the `TwilightBus` class works from the perspective of a client of the class.

Example 6-14. Passengers disappear when dropped by a TwilightBus

```
>>> basketball_team = ['Sue', 'Tina', 'Maya', 'Diana', 'Pat'] ❶
>>> bus = TwilightBus(basketball_team) ❷
>>> bus.drop('Tina') ❸
>>> bus.drop('Pat')
>>> basketball_team ❹
['Sue', 'Maya', 'Diana']
```

- 
- ❶ `basketball_team` holds five student names.
 - ❷ A `TwilightBus` is loaded with the team.
 - ❸ The `bus` drops one student, then another.
 - ❹ The dropped passengers vanished from the basketball team!

`TwilightBus` violates the “Principle of least astonishment,” a best practice of interface design. It surely is astonishing that when the bus drops a student, her name is removed from the basketball team roster.

Example 6-15 is the implementation `TwilightBus` and an explanation of the problem.

Example 6-15. A simple class to show the perils of mutating received arguments

```
class TwilightBus:  
    """A bus model that makes passengers vanish"""  
  
    def __init__(self, passengers=None):  
        if passengers is None:  
            self.passengers = [] ❶  
        else:  
            self.passengers = passengers ❷  
  
    def pick(self, name):  
        self.passengers.append(name)  
  
    def drop(self, name):  
        self.passengers.remove(name) ❸
```



- ❶ Here we are careful to create a new empty list when `passengers` is `None`.
- ❷ However, this assignment makes `self.passengers` an alias for `passengers`, which is itself an alias for the actual argument passed to `__init__` (i.e., `basketball_team` in Example 6-14).
- ❸ When the methods `.remove()` and `.append()` are used with `self.passengers`, we are actually mutating the original list received as argument to the constructor.

The problem here is that the bus is aliasing the list that is passed to the constructor. Instead, it should keep its own passenger list. The fix is simple: in `__init__`, when the `passengers` parameter is provided,

`self.passengers` should be initialized with a copy of it, as we did correctly in [Example 6-8 \(“Deep and Shallow Copies of Arbitrary Objects”\)](#):

```
def __init__(self, passengers=None):
    if passengers is None:
        self.passengers = []
    else:
        self.passengers = list(passengers) ❶
```

- ❶ Make a copy of the `passengers` list, or convert it to a `list` if it's not one.

Now our internal handling of the passenger list will not affect the argument used to initialize the bus. As a bonus, this solution is more flexible: now the argument passed to the `passengers` parameter may be a `tuple` or any other iterable, like a `set` or even database results, because the `list` constructor accepts any iterable. As we create our own list to manage, we ensure that it supports the necessary `.remove()` and `.append()` operations we use in the `.pick()` and `.drop()` methods.

TIP

Unless a method is explicitly intended to mutate an object received as argument, you should think twice before aliasing the argument object by simply assigning it to an instance variable in your class. If in doubt, make a copy. Your clients will often be happier.

del and Garbage Collection

Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected.

—Data Model, chapter of *The Python Language Reference*

The `del` statement deletes names, not objects. An object may be garbage collected as result of a `del` command, but only if the variable deleted holds the last reference to the object, or if the object becomes unreachable.² Rebinding a variable may also cause the number of references to an object to reach zero, causing its destruction.

WARNING

There is a `__del__` special method, but it does not cause the disposal of the instance, and should not be called by your code. `__del__` is invoked by the Python interpreter when the instance is about to be destroyed to give it a chance to release external resources. You will seldom need to implement `__del__` in your own code, yet some Python beginners spend time coding it for no good reason. The proper use of `__del__` is rather tricky. See the [__del__ special method documentation](#) in the “Data Model” chapter of *The Python Language Reference*.

In CPython, the primary algorithm for garbage collection is reference counting. Essentially, each object keeps count of how many references point to it. As soon as that *refcount* reaches zero, the object is immediately destroyed: CPython calls the `__del__` method on the object (if defined) and then frees the memory allocated to the object. In CPython 2.0, a generational garbage collection algorithm was added to detect groups of objects involved in reference cycles—which may be unreachable even with outstanding references to them, when all the mutual references are contained within the group. Other implementations of Python have more sophisticated garbage collectors that do not rely on reference counting, which means the `__del__` method may not be called immediately when there are no more references to the object. See [“PyPy, Garbage Collection,](#)

[and a Deadlock](#)” by A. Jesse Jiryu Davis for discussion of improper and proper use of `__del__`.

To demonstrate the end of an object’s life, [Example 6-16](#) uses `weakref.finalize` to register a callback function to be called when an object is destroyed.

Example 6-16. Watching the end of an object when no more references point to it

```
>>> import weakref
>>> s1 = {1, 2, 3}          ❶
>>> s2 = s1
>>> def bye():            ❷
...     print('Gone with the wind...')
...
>>> ender = weakref.finalize(s1, bye) ❸
>>> ender.alive ❹
True
>>> del s1
>>> ender.alive ❺
True
>>> s2 = 'spam' ❻
Gone with the wind...
>>> ender.alive
False
```

- ❶ `s1` and `s2` are aliases referring to the same set, `{1, 2, 3}`.
- ❷ This function must not be a bound method of the object about to be destroyed or otherwise hold a reference to it.
- ❸ Register the `bye` callback on the object referred by `s1`.
- ❹ The `.alive` attribute is `True` before the `finalize` object is called.
- ❺ As discussed, `del` does not delete an object, just a reference to it.
- ❻ Rebinding the last reference, `s2`, makes `{1, 2, 3}` unreachable. It is destroyed, the `bye` callback is invoked, and `ender.alive` becomes `False`.

The point of [Example 6-16](#) is to make explicit that `del` does not delete

objects, but objects may be deleted as a consequence of being unreachable after `del` is used.

You may be wondering why the `{1, 2, 3}` object was destroyed in [Example 6-16](#). After all, the `s1` reference was passed to the `finalize` function, which must have held on to it in order to monitor the object and invoke the callback. This works because `finalize` holds a *weak reference* to `{1, 2, 3}`, as explained in the next section.

Weak References

The presence of references is what keeps an object alive in memory. When the reference count of an object reaches zero, the garbage collector disposes of it. But sometimes it is useful to have a reference to an object that does not keep it around longer than necessary. A common use case is a cache.

Weak references to an object do not increase its reference count. The object that is the target of a reference is called the *referent*. Therefore, we say that a weak reference does not prevent the referent from being garbage collected.

Weak references are useful in caching applications because you don't want the cached objects to be kept alive just because they are referenced by the cache.

[Example 6-17](#) shows how a `weakref.ref` instance can be called to reach its referent. If the object is alive, calling the weak reference returns it, otherwise `None` is returned.

TIP

Example 6-17 is a console session, and the Python console automatically binds the `_` variable to the result of expressions that are not `None`. This interfered with my intended demonstration but also highlights a practical matter: when trying to micro-manage memory we are often surprised by hidden, implicit assignments that create new references to our objects. The `_` console variable is one example. Traceback objects are another common source of unexpected references.

Example 6-17. A weak reference is a callable that returns the referenced object or `None` if the referent is no more

```
>>> import weakref
>>> a_set = {0, 1}
>>> wref = weakref.ref(a_set) ❶
>>> wref
<weakref at 0x100637598; to 'set' at 0x100636748>
>>> wref() ❷
{0, 1}
>>> a_set = {2, 3, 4} ❸
>>> wref() ❹
{0, 1}
>>> wref() is None ❺
False
>>> wref() is None ❻
True
```

-
- ❶ The `wref` weak reference object is created and inspected in the next line.
 - ❷ Invoking `wref()` returns the referenced object, `{0, 1}`. Because this is a console session, the result `{0, 1}` is bound to the `_` variable.
 - ❸ `a_set` no longer refers to the `{0, 1}` set, so its reference count is decreased. But the `_` variable still refers to it.
 - ❹ Calling `wref()` still returns `{0, 1}`.
 - ❺ When this expression is evaluated, `{0, 1}` lives, therefore `wref()` is not `None`. But `_` is then bound to the resulting value, `False`. Now there are no more strong references to `{0, 1}`.
 - ❻ When this expression is evaluated, `{0, 1}` has been garbage collected, so `wref()` returns `None`.

- ❶ Because the `{0, 1}` object is now gone, this last call to `wref()` returns `None`.

The [weakref module documentation](#) makes the point that the `weakref.ref` class is actually a low-level interface intended for advanced uses, and that most programs are better served by the use of the `weakref` collections and `finalize`. In other words, consider using `WeakKeyDictionary`, `WeakValueDictionary`, `WeakSet`, and `finalize` (which use weak references internally) instead of creating and handling your own `weakref.ref` instances by hand. We just did that in [Example 6-17](#) in the hope that showing a single `weakref.ref` in action could take away some of the mystery around them. But in practice, most of the time Python programs use the `weakref` collections.

The next subsection briefly discusses the `weakref` collections.

The WeakValueDictionary Skit

The class `WeakValueDictionary` implements a mutable mapping where the values are weak references to objects. When a referred object is garbage collected elsewhere in the program, the corresponding key is automatically removed from `WeakValueDictionary`. This is commonly used for caching.

Our demonstration of a `WeakValueDictionary` is inspired by the classic *Cheese Shop* skit by Monty Python, in which a customer asks for more than 40 kinds of cheese, including cheddar and mozzarella, but none are in stock.³

[Example 6-18](#) implements a trivial class to represent each kind of cheese.

Example 6-18. Cheese has a kind attribute and a standard representation

```
class Cheese:  
  
    def __init__(self, kind):  
        self.kind = kind  
  
    def __repr__(self):  
        return 'Cheese(%r)' % self.kind
```

In Example 6-19, each cheese is loaded from a `catalog` to a `stock` implemented as a `WeakValueDictionary`. However, all but one disappear from the `stock` as soon as the `catalog` is deleted. Can you explain why the Parmesan cheese lasts longer than the others?⁴ The tip after the code has the answer.

Example 6-19. Customer: “Have you in fact got any cheese here at all?”

```
>>> import weakref  
>>> stock = weakref.WeakValueDictionary() ❶  
>>> catalog = [Cheese('Red Leicester'), Cheese('Tilsit'),  
...                 Cheese('Brie'), Cheese('Parmesan')]  
...  
>>> for cheese in catalog:  
...     stock[cheese.kind] = cheese ❷  
...  
>>> sorted(stock.keys())  
['Brie', 'Parmesan', 'Red Leicester', 'Tilsit'] ❸  
>>> del catalog  
>>> sorted(stock.keys())  
['Parmesan'] ❹  
>>> del cheese  
>>> sorted(stock.keys())  
[]
```

- ❶ `stock` is a `WeakValueDictionary`.
- ❷ The `stock` maps the name of the cheese to a weak reference to the cheese instance in the `catalog`.
- ❸ The `stock` is complete.
- ❹ After the `catalog` is deleted, most cheeses are gone from the

`stock`, as expected in `WeakValueDictionary`. Why not all, in this case?

TIP

A temporary variable may cause an object to last longer than expected by holding a reference to it. This is usually not a problem with local variables: they are destroyed when the function returns. But in [Example 6-19](#), the `for` loop variable `cheese` is a global variable and will never go away unless explicitly deleted.

A counterpart to the `WeakValueDictionary` is the `WeakKeyDictionary` in which the keys are weak references. The [`weakref.WeakKeyDictionary`](#) documentation hints on possible uses:

[A `WeakKeyDictionary`] can be used to associate additional data with an object owned by other parts of an application without adding attributes to those objects. This can be especially useful with objects that override attribute accesses.

The `weakref` module also provides a `WeakSet`, simply described in the docs as “Set class that keeps weak references to its elements. An element will be discarded when no strong reference to it exists any more.” If you need to build a class that is aware of every one of its instances, a good solution is to create a class attribute with a `WeakSet` to hold the references to the instances. Otherwise, if a regular `set` was used, the instances would never be garbage collected, because the class itself would have strong references to them, and classes live as long as the Python process unless you deliberately delete them.

These collections, and weak references in general, are limited in the kinds

of objects they can handle. The next section explains.

Limitations of Weak References

Not every Python object may be the target, or referent, of a weak reference. Basic `list` and `dict` instances may not be referents, but a plain subclass of either can solve this problem easily:

```
class MyList(list):
    """list subclass whose instances may be weakly referenced"""

a_list = MyList(range(10))

# a_list can be the target of a weak reference
wref_to_a_list = weakref.ref(a_list)
```

A `set` instance can be a referent, and that's why a `set` was used in [Example 6-17](#). User-defined types also pose no problem, which explains why the silly `Cheese` class was needed in [Example 6-19](#). But `int` and `tuple` instances cannot be targets of weak references, even if subclasses of those types are created.

Most of these limitations are implementation details of CPython that may not apply to other Python interpreters. They are the result of internal optimizations, some of which are discussed in the following (highly optional) section.

Tricks Python Plays with Immutables

NOTE

This optional section discusses some Python details that are not really important for users of Python, and that may not apply to other Python implementations or even

future versions of CPython. Nevertheless, I've seen people stumble upon these corner cases and then start using the `is` operator incorrectly, so I felt they were worth mentioning.

I was surprised to learn that, for a tuple `t`, `t[:]` does not make a copy, but returns a reference to the same object. You also get a reference to the same tuple if you write `tuple(t)`.⁵ [Example 6-20](#) proves it.

Example 6-20. A tuple built from another is actually the same exact tuple

```
>>> t1 = (1, 2, 3)
>>> t2 = tuple(t1)
>>> t2 is t1 ❶
True
>>> t3 = t1[:]
>>> t3 is t1 ❷
True
```

- ❶ `t1` and `t2` are bound to the same object.
❷ And so is `t3`.

The same behavior can be observed with instances of `str`, `bytes`, and `frozenset`. Note that a `frozenset` is not a sequence, so `fs[:]` does not work if `fs` is a `frozenset`. But `fs.copy()` has the same effect: it cheats and returns a reference to the same object, and not a copy at all, as [Example 6-21](#) shows.⁶

Example 6-21. String literals may create shared objects

```
>>> t1 = (1, 2, 3)
>>> t3 = (1, 2, 3) ❶
>>> t3 is t1 ❷
False
>>> s1 = 'ABC'
>>> s2 = 'ABC' ❸
>>> s2 is s1 ❹
True
```

- ❶ Creating a new tuple from scratch.
- ❷ `t1` and `t3` are equal, but not the same object.
- ❸ Creating a second `str` from scratch.
- ❹ Surprise: `a` and `b` refer to the same `str`!

The sharing of string literals is an optimization technique called *interning*. CPython uses the same technique with small integers to avoid unnecessary duplication of numbers that appear frequently in programs like 0, 1, -1, etc. Note that CPython does not intern all strings or integers, and the criteria it uses to do so is an undocumented implementation detail.

WARNING

Never depend on `str` or `int` interning! Always use `==` and not `is` to compare them for equality. Interning is an optimization for internal use of the Python interpreter.

The tricks discussed in this section, including the behavior of `frozenset.copy()`, are harmless “lies”; they save memory and make the interpreter faster. Do not worry about them, they should not give you any trouble because they only apply to immutable types. Probably the best use of these bits of trivia is to win bets with fellow Pythonistas.

Chapter Summary

Every Python object has an identity, a type, and a value. Only the value of an object changes over time.⁷

If two variables refer to immutable objects that have equal values (`a == b` is `True`), in practice it rarely matters if they refer to copies or are aliases referring to the same object because the value of an immutable object does not change, with one exception. The exception is immutable collections such as tuples and frozensets: if an immutable collection holds references to mutable items, then its value may actually change when the value of a mutable item changes. In practice, this scenario is not so common. What never changes in an immutable collection are the identities of the objects within.

The fact that variables hold references has many practical consequences in Python programming:

- Simple assignment does not create copies.
- Augmented assignment with `+=` or `*=` creates new objects if the lefthand variable is bound to an immutable object, but may modify a mutable object in place.
- Assigning a new value to an existing variable does not change the object previously bound to it. This is called a rebinding: the variable is now bound to a different object. If that variable was the last reference to the previous object, that object will be garbage collected.
- Function parameters are passed as aliases, which means the function may change any mutable object received as an argument.

There is no way to prevent this, except making local copies or using immutable objects (e.g., passing a tuple instead of a list).

- Using mutable objects as default values for function parameters is dangerous because if the parameters are changed in place, then the default is changed, affecting every future call that relies on the default.

In CPython, objects are discarded as soon as the number of references to them reaches zero. They may also be discarded if they form groups with cyclic references but no outside references. In some situations, it may be useful to hold a reference to an object that will not—by itself—keep an object alive. One example is a class that wants to keep track of all its current instances. This can be done with weak references, a low-level mechanism underlying the more useful collections `WeakValueDictionary`, `WeakKeyDictionary`, `WeakSet`, and the `finalize` function from the `weakref` module.

Further Reading

The “[Data Model](#)” chapter of *The Python Language Reference* starts with a clear explanation of object identities and values.

Wesley Chun, author of the *Core Python* series of books, made a great presentation about many of the topics covered in this chapter during OSCON 2013. You can download the slides from the “[Python 103: Memory Model & Best Practices](#)” talk page. There is also a [YouTube video](#) of a longer presentation Wesley gave at EuroPython 2011, covering not only the theme of this chapter but also the use of special methods.

Doug Hellmann wrote a long series of excellent blog posts titled [Python Module of the Week](#), which became a book, [The Python Standard Library](#).

by Example. His posts “[copy – Duplicate Objects](#)” and “[weakref – Garbage-Collectable References to Objects](#)” cover some of the topics we just discussed.

More information on the CPython generational garbage collector can be found in the [gc module documentation](#), which starts with the sentence “This module provides an interface to the optional garbage collector.” The “optional” qualifier here may be surprising, but the [“Data Model” chapter](#) also states:

An implementation is allowed to postpone garbage collection or omit it altogether—it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable.

Fredrik Lundh—creator of key libraries like ElementTree, Tkinter, and the PIL image library—has a short post about the Python garbage collector titled [“How Does Python Manage Memory?”](#) He emphasizes that the garbage collector is an implementation feature that behaves differently across Python interpreters. For example, Jython uses the Java garbage collector.

The CPython 3.4 garbage collector improved handling of objects with a `__del__` method, as described in [PEP 442 — Safe object finalization](#).

Wikipedia has an article about [string interning](#), mentioning the use of this technique in several languages, including Python.

SOAPBOX

Equal Treatment to All Objects

I learned Java before I discovered Python. The `==` operator in Java never felt right for me. It is much more common for programmers to care about equality than identity, but for objects (not primitive types) the Java

```
== compares references, and not object values. Even for something as basic as comparing strings, Java forces you to use the .equals method. Even then, there is another catch: if you write a.equals(b) and a is null, you get a null pointer exception. The Java designers felt the need to overload + for strings, so why not go ahead and overload == as well?
```

Python gets this right. The == operator compares object values and is compares references. And because Python has operator overloading, == works sensibly with all objects in the standard library, including None, which is a proper object, unlike Java's null.

And of course, you can define __eq__ in your own classes to decide what == means for your instances. If you don't override __eq__, the method inherited from object compares object IDs, so the fallback is that every instance of a user-defined class is considered different.

These are some of the things that made me switch from Java to Python as soon as I finished reading the Python Tutorial one afternoon in September 1998.

Mutability

This chapter would be redundant if all Python objects were immutable. When you are dealing with unchanging objects, it makes no difference whether variables hold the actual objects or references to shared objects. If a == b is true, and neither object can change, they might as well be the same. That's why string interning is safe. Object identity becomes important only when objects are mutable.

In "pure" functional programming, all data is immutable: appending to a collection actually creates a new collection. Python, however, is not a functional language, much less a pure one. Instances of user-defined classes are mutable by default in Python—as in most object-oriented languages. When creating your own objects, you have to be extra careful to make them immutable, if that is a requirement. Every attribute of the object must also be immutable, otherwise you end up with something like the tuple: immutable as far as object IDs go, but the value of a tuple may change if it holds a mutable object.

Mutable objects are also the main reason why programming with threads is so hard to get right: threads mutating objects without proper synchronization produce corrupted data. Excessive synchronization, on the other hand, causes deadlocks.

Object Destruction and Garbage Collection

There is no mechanism in Python to directly destroy an object, and this omission is actually a great feature: if you could destroy an object at any time, what would happen to existing strong references pointing to it?

Garbage collection in CPython is done primarily by reference counting, which is easy to implement, but is prone to memory leaking when there are reference cycles, so with version 2.0 (October 2000) a generational garbage collector was implemented, and it is able to dispose of unreachable objects kept alive by reference cycles.

But the reference counting is still there as a baseline, and it causes the immediate disposal of objects with zero references. This means that, in CPython—at least for now—it's safe to write this:

```
open('test.txt', 'wt', encoding='utf-8').write('1, 2, 3')
```

That code is safe because the reference count of the file object will be zero after the write method returns, and Python will immediately close the file before destroying the object representing it in memory.

However, the same line is not safe in Jython or IronPython that use the garbage collector of their host runtimes (the Java VM and the .NET CLR), which are more sophisticated but do not rely on reference counting and may take longer to destroy the object and close the file. In all cases, including CPython, the best practice is to explicitly close the file, and the most reliable way of doing it is using the `with` statement, which guarantees that the file will be closed even if exceptions are raised while it is open. Using `with`, the previous snippet becomes:

```
with open('test.txt', 'wt', encoding='utf-8') as fp:  
    fp.write('1, 2, 3')
```

If you are into the subject of garbage collectors, you may want to read Thomas Perl's paper "[Python Garbage Collector Implementations: CPython, PyPy and GaS](#)", from which I learned the bit about the safety of the `open().write()` in CPython.

Parameter Passing: Call by Sharing

A popular way of explaining how parameter passing works in Python is the phrase: "Parameters are passed by value, but the values are references." This is not wrong, but causes confusion because the most common parameter passing modes in older languages are *call by value* (the function gets a copy of the argument) and *call by reference* (the function gets a pointer to the argument). In Python, the function gets a copy of the arguments, but the arguments are always references. So the value of the referenced objects may be changed, if they are mutable, but their identity cannot. Also, because the function gets a copy of the reference in an argument, rebinding it has no effect outside of the function. I adopted the term *call by sharing* after reading up on the subject in *Programming Language Pragmatics, Third Edition* by Michael L. Scott (Morgan Kaufmann), particularly "8.3.1: Parameter Modes."

The Full Quote of Alice and the Knights's Song

I love this passage, but it was too long as a chapter opener. So here is the complete dialog about the Knight's song, its name, and how the song and its name are called:

'You are sad,' the Knight said in an anxious tone: 'let me sing you a song to comfort you.'

'Is it very long?' Alice asked, for she had heard a good deal of poetry that day.

'It's long,' said the Knight, 'but very, VERY beautiful. Everybody that hears me sing it—either it brings the TEARS into their eyes, or else—'

'Or else what?' said Alice, for the Knight had made a sudden pause.

'Or else it doesn't, you know. The name of the song is called "HADDOCKS' EYES".'

'Oh, that's the name of the song, is it?' Alice said, trying to feel interested.

'No, you don't understand,' the Knight said, looking a little vexed. 'That's what the name is CALLED. The name really IS "THE AGED AGED MAN".'

'Then I ought to have said "That's what the SONG is called"?' Alice corrected herself.

'No, you oughtn't: that's quite another thing! The SONG is called "WAYS AND MEANS": but that's only what it's CALLED, you know!'

'Well, what IS the song, then?' said Alice, who was by this time completely bewildered.

'I was coming to that,' the Knight said. 'The song really IS "A-SITTING ON A GATE": and the tune's my own invention.'

—Lewis Carroll, Through the Looking-Glass, Chapter VIII, "It's My Own Invention"

-
- On the other hand, flat sequences like `str`, `bytes`, and `array.array` don't contain references but physically hold their data—characters, bytes, and numbers—in contiguous memory.
- If two objects refer to each other, as in [Example 6-10](#), they may be destroyed if the garbage collector determines that they are otherwise unreachable because their only references are their mutual references.
- `cheeseshop.python.org` is also an alias for PyPI—the Python Package Index software repository—which started its life quite empty. At the time of this writing, the Python Cheese Shop has 41,426 packages. Not bad, but still far from the more than 131,000 modules available in CPAN—the Comprehensive Perl Archive Network—the envy of all dynamic language communities.
- Parmesan cheese is aged at least a year at the factory, so it is more durable than fresh cheese, but this is not the answer we are looking for.
- This is clearly documented. Type `help(tuple)` in the Python console to read: “If the argument is a tuple, the return value is the same object.” I thought I knew everything about tuples before writing this book.
- The harmless lie of having the `copy` method not copying anything can be justified by interface compatibility: it makes `frozenset` more compatible with `set`. Anyway, it makes no difference to the end user whether two identical immutable objects are the same or are copies.
- Actually the type of an object may be changed by merely assigning a different class to its `__class__` attribute, but that is pure evil and I regret writing this footnote.