
An array of sequences

As you may have noticed, several of the operations mentioned work equally for texts, lists and tables. Texts, lists and tables together are called *trains*. [...] The FOR command also works generically on trains¹.

— 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 sequences, built-in tuple and mapping types, structure by indentation, strong typing without variable declarations etc. 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 focus on Unicode strings and byte sequences is deferred to [Chapter 4](#). Also, the idea here is to cover sequence types that are ready to use. Creating your own sequence types is the subject of [Chapter 10](#).

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

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.

Flat sequences

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

Container sequences hold references to the objects they contain, which may be of any type, while *flat sequences* physically store the value of each item within its own memory space, and not as distinct objects. Thus, flat sequences are more compact, but they are limited to holding primitive values like characters, bytes and numbers

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`

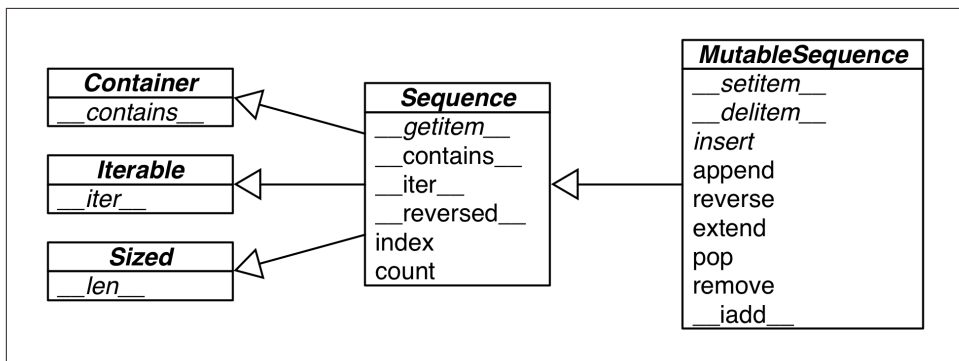


Figure 2-1. UML class diagram for some classes from `collections.abc`. Superclasses are on the left; inheritance arrows point from subclasses to superclasses. Names in *italic* are abstract classes and abstract methods.

Figure 2-1 helps visualize how mutable sequences differ from immutable ones, while also inheriting several methods from them. Note that the built-in concrete sequence types do not actually subclass the `Sequence` and `MutableSequence` ABCs (Abstract Base Classes) depicted, but the ABCs are still useful as a formalization of what functionality to expect from a full-featured sequence type.

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. The code in [Example 2-1](#) is building up a list. In contrast, a listcomp is meant to do one thing only: 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. Please don't use that syntax if you are not doing something with the produced list. 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 multi-line lists, listcomps, genexps, dictionaries etc. without using the ugly `\` line continuation escape.

Listcomps no longer leak their variables

In Python 2.x, variables assigned in the for clauses in list comprehensions were set in the surrounding scope, sometimes with tragic consequences. See the following Python 2.7 console session:

```
Python 2.7.6 (default, Mar 22 2014, 22:59:38)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 'my precious'
>>> dummy = [x for x in 'ABC']
>>> x
'C'
```

As you can see, the initial value of `x` was clobbered. This no longer happens in Python 3.

List comprehensions, generator expressions and their siblings set and dict comprehensions now 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.

This is Python 3:

```
>>> x = 'ABC'
>>> dummy = [ord(x) for x in x]
>>> x ❶
```

```
'ABC'
>>> dummy ❷
[65, 66, 67]
>>>
```

- ❶ The value of `x` is preserved.
- ❷ The list comprehension produces the expected list.

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 = '$¢£¥€π'
>>> 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 examples above².

I'll have more to say about `map` and `filter` in [Chapter 5](#). Now we turn to the use of listcomps to compute cartesian products: a list containing tuples build from all items from two or more lists.

Cartesian products

Listcomps can generate 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 2-2](#).

2. The `02-array-seq/listcomp_speed.py` script in the Fluent Python code repository is a simple speed test comparing listcomp with `filter/map`.

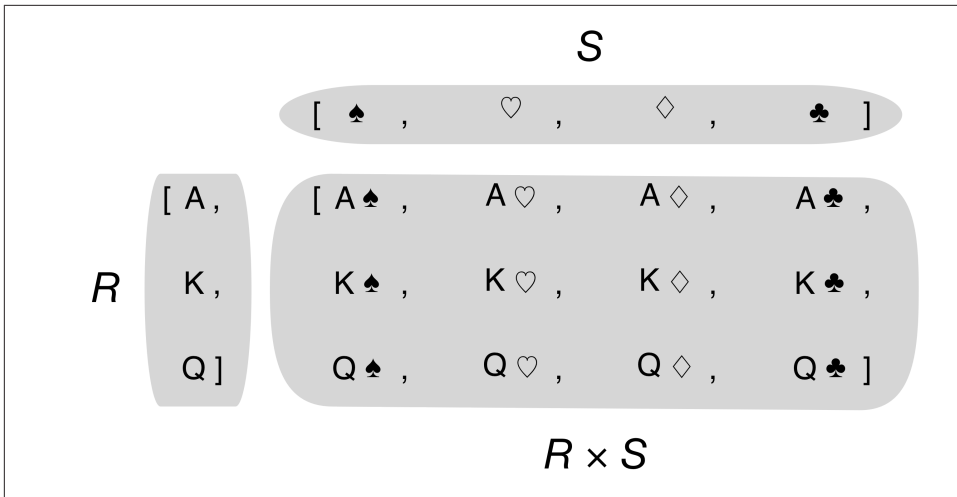


Figure 2-2. The cartesian product of a sequence of three card ranks and a sequence of four suits results in a sequence of twelve pairings.

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 6 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 suits and 4 ranks, grouped by suit:

```
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 fill-up other sequence types, a genexp is the way to go. The next section is a brief look at genexps in the context of building non-list 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 parenthesis 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 parenthesis.
- ❷ The array constructor takes two arguments, so the parenthesis around the generator expression are mandatory³.

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 6-item list of t-shirts is never built in memory: the generator expression feeds the for loop producing one item at a time. If the two lists used in the cartesian product had a thousand items each, using

3. The first argument of the array constructor defines the storage type used for the numbers in the array, as we'll see in [“Arrays” on page 48](#)

a generator expression would save the expense 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 6 t-shirt variations is never produced in this example.

Chapter 14 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 often fixed and their order is always vital.

Example 2-7 shows tuples being used as records. Note that in every expression below, 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, 32450, 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 (millions), 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 tuple unpacking mechanism — our next subject.

Tuple unpacking

In [Example 2-7](#) we assigned `('Tokyo', 2003, 32450, 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*.



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” on page 29. 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 tuple 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 # tuple 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 tuple 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 code above also shows a further use of tuple unpacking: enabling functions to return multiple values in a way that is convenient to the caller. For 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 example above.



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 nice name for placeholder variable.

Another way of focusing on just some of the items when unpacking a tuple is to use the `*`, 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. 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 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



Before Python 3, it used to be possible to define functions with nested tuples in the formal parameters, eg. `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.

As designed, tuples are very handy. But there is a missing feature when using them as records: sometimes it is desirable to name the fields. That is why the `namedtuple` function was invented. Read on.

Named tuples

The `collections.namedtuple` function is a factory that produces subclasses of `tuple` enhanced with field names and a class name — which helps debugging.



Instances of a class that you build with `namedtuple` take exactly the same amount of memory as tuples because the field names are stored in the class. They use less memory than a regular object because they do store attributes in a per-instance `__dict__`.

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

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

Example 2-9 shows how we could define a named tuple to hold information about a city:

Example 2-9. 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 Data must be passed as positional arguments to the constructor (in contrast, the tuple constructor takes a single iterable).
- ❸ You can access the fields by name or position.

A named tuple type has a few attributes in addition to those inherited from tuple.

Example 2-10 shows the most useful: the `_fields` class attribute, the class method `_make(iterable)` and the `_asdict()` instance method.

*Example 2-10. Named tuple attributes and methods (continued from **Example 2-9**)*

```
>>> City._fields ❶
('name', 'country', 'population', 'coordinates')
>>> LatLong = namedtuple('LatLong', 'lat long')
>>> delhi_data = ('Delhi NCR', 'IN', 21.935, LatLong(28.613889, 77.208889))
>>> delhi = City._make(delhi_data) ❷
>>> delhi._asdict() ❸
OrderedDict([('name', 'Delhi NCR'), ('country', 'IN'), ('population',
21.935), ('coordinates', LatLong(lat=28.613889, long=77.208889))])
>>> for key, value in delhi._asdict().items():
    print(key + ': ', value)

name: Delhi NCR
country: IN
population: 21.935
coordinates: LatLong(lat=28.613889, long=77.208889)
>>>
```

- ❶ `_fields` is a tuple with the field names of the class.
- ❷ `_make()` lets you instantiate a named tuple from an iterable; `City(*delhi_data)` would do the same.
- ❸ `_asdict()` returns a `collections.OrderedDict` built from the named tuple instance. That can be used to produce a nice display of city data.

Now that we’ve explored the power of tuples as records, we can consider their second role as an immutable variant of the `list` type.

Tuples as immutable lists

When using a tuple as an immutable variation of `list`, it helps to know how similar they actually are. As you can see in [Table 2-1](#), all `list` methods that do not involve adding or removing items are supported by tuple with one exception — tuple lacks the `__reversed__` method, but 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 ³

	list	tuple
<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 [Chapter 13](#).

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

Slicing

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

In this section we describe the *use* of these advanced forms of slicing. Their implementation in a user-defined class will be covered in [Chapter 10](#), in keeping with our philosophy of covering ready-to-use classes in this part of the book, and creating new classes in Part IV.

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:]`. See 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. Please see the last reference in “[Further reading](#)” on page 58.

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

Slice objects

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

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

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

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

The notation `a:b:c` is only valid within `[]` when used as the indexing or subscript operator, and it produces a slice object: `slice(a, b, c)`. As we will see in “[How slicing works](#)” on page 283, 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 on top of [Example 2-11](#). Instead of filling your code with hard-coded slices, you can name them. See how readable this makes the for loop at the end of the example:

Example 2-11. 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

```

When we discuss creating your own collections in “[Vector take #2: a sliceable sequence](#)” on page 282, we’ll come back to slice objects. Meanwhile, from a user perspective slicing includes additional features such as multi-dimensional slices and ellipsis (...) notation. Read on.

Multi-dimensional slicing and ellipsis

The `[]` operator can also take multiple indexes or slices separated by commas. This is used, for instance, in the external NumPy package, where items of a 2-dimensional `numpy.ndarray` can be fetched using the syntax `a[i, j]` and a 2-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. The `__getitem__` and `__setitem__` special methods which 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))`.

The built-in sequence types in Python are one-dimensional, so they support only one index or slice, and not a tuple of them.

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` class⁴. 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 4-dimensional array, `x[i, ...]` is a shortcut for `x[i, :, :, :]`. See the [Tentative NumPy Tutorial](#) to learn more about this.

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

4. 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`.

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-hand side must be an iterable object, even if it has just one item.

Everybody knows that concatenation is a common operation with sequences of any type. Any introductory Python text explains 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 the 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'
'abcdabcdabcdabcd'
```

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



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

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

Building lists of lists

Sometimes we need to initialize a list with a certain number of nested lists, for example, to distribute students in a list of teams or to represent squares on a game board. The best way of doing so is with a list comprehension, like this:

Example 2-12. A list with 3 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 with 3 lists of 3 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 2-13](#)

Example 2-13. A list with 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 2-13](#) 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 3 times to board.

On the other hand, the list comprehension from [Example 2-12](#) is equivalent to this code:

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

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



If either the problem or the solution in this section are not clear to you, relax. The entire [Chapter 8](#) 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 very 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 like `list`, `bytearray`, `array.array`, `a` will be changed in-place, i.e. the effect will be similar to `a.extend(b)`. However, when `a` does not implement `__iadd__`, the expression `a += b` has the same effect as `a = a + b`: the expression `a + b` is evaluated first, producing a new object which is then bound to `a`. In other words, the identity of the object bound to `a` may or may not change, depending on the availability of `__iadd__`.

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

What I just wrote about `+=` also applies to `*=`, which is implemented via `__imul__`. The `__iadd__` and `__imul__` special methods are discussed in [Chapter 13](#).

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 concatenated⁵.

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.

5. `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.

A += assignment puzzler

Try to answer without using the console: what is the result of evaluating these two expressions⁶?

Example 2-14. 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**.”! Here is the actual output from a Python 3.4 console (actually the result is the same in a Python 2.7 console):

Example 2-15. The unexpected result: item `t[2]` is changed and an exception is raised⁷.

```
>>> 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 2-3](#) is a composite of two screenshots showing the initial and final states of the tuple `t` from [Example 2-15](#).

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

7. A reader suggested that the operation in the example can be performed with `t[2].extend([50, 60])`, without errors. We're aware of that, but the intent of the example is to discuss the odd behavior of the `+=` operator.

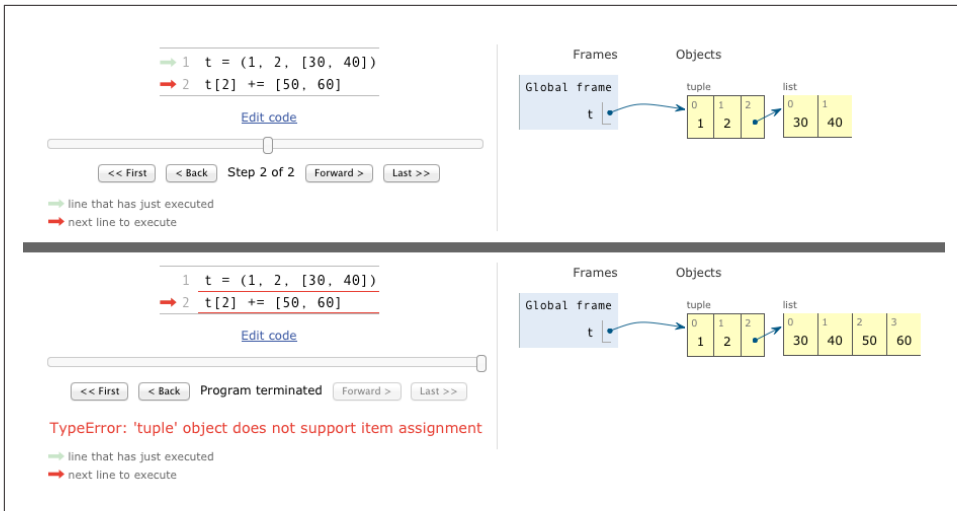


Figure 2-3. Initial and final state of the tuple assignment puzzler. (Diagram generated by *Online Python Tutor*)

If you look at the bytecode Python generates for the expression `s[a] += b` (Example 2-16), it becomes clear how that happens.

Example 2-16. Bytecode for the expression `s[a] += b`.

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

- ❶ Put the value of `s[a]` on TOS (Top Of Stack).
- ❷ Perform `TOS += b`. This succeeds if TOS refers to a mutable object (it's a list, in Example 2-15).
- ❸ Assign `s[a] = TOS`. This fails if `s` is immutable (the `t` tuple in Example 2-15).

This example is quite a corner case — in 15 years of using Python I have never seen this strange behavior actually bite somebody.

I take three lessons from this:

- Putting mutable items in tuples is not a good idea.
- 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 is often 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 target object, 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 object itself was changed, and no new object was created. The same behavior can be seen, for example, in the `random.shuffle` function.



The convention of returning `None` to signal in-place changes has a drawback: you cannot cascade calls to those methods. In contrast, methods that return new objects (eg. all `str` methods) can be cascaded in the fluent interface style. See [Fluent interface](#) on Wikipedia for a description of fluent interfaces.

In contrast, the built-in function `sorted` creates a new list and returns it. In fact, `sorted` accepts any iterable object as argument, including immutable sequences and generators (see [Chapter 14](#)). 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: `key` and `reverse`.

`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.



The key optional keyword parameter can also be used with the `min()` and `max()` built-ins and with other functions from the standard library such as `itertools.groupby()` and `heapq.nlargest()`.

Here are a few examples to clarify the use of these functions and keyword arguments. The examples also demonstrate that Timsort — the sorting algorithm used in Python — is stable, i.e. it preserves the relative ordering of items that compare equal⁸.

```
>>> 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 alphabetically.
- ❷ Inspecting the original list, we see it is unchanged.
- ❸ This is simply reverse alphabetical ordering.
- ❹ A new list of strings, now sorted by length. Since 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.

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

8. The **Soapbox** at the end of this chapter has more about Timsort.

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 or equal to `needle`. You could use the result of `bisect(haystack, needle)` as the index argument to `haystack.insert(index, needle)`, but using `insort` does both steps, and is faster.



Raymond Hettinger — a prolific Python contributor — has a **Sorted Collection recipe** which leverages the `bisect` module but is easier to use than these stand-alone functions.

Example 2-17 uses a carefully chosen set of “needles” to demonstrate the insert positions returned by `bisect`. Its output is in **Figure 2-4**.

Example 2-17. `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__) ❸
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        | | | | | 8
 5 @ 3        | | 15
 2 @ 1        | 2
 1 @ 1        | 1
 0 @ 0        0

```

Figure 2-4. Output of *Example 2-17* 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* this 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*.

```

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 | 5
2 @ 1 | 2
1 @ 0 1
0 @ 0 0

```

Figure 2-5. Output of [Example 2-17](#) with `bisect_left` in use. Compare with [Figure 2-4](#) 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 2-18](#).

Example 2-18. 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 [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']

```

The code in [Example 2-18](#) 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.

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 2-19](#) and its output in [Figure 2-6](#).

Example 2-19. `insort` 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_insort.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 2-6. Output of *Example 2-19*.

Like `bisect`, `insort` takes optional `lo`, `hi` arguments to limit the search to a subsequence. There is also an `insort_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 to them.

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, if you need to store 10 million of floating point values an array is much more efficient, because an array does not actually hold full-fledged float objects, but only the packed bytes representing their machine values — just like an array in the C language. On the other hand, if you are constantly adding and removing items from the ends of a list as a FIFO or LIFO data structure, a deque (double-ended queue) works faster.



If your code does a lot of containment checks — 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 all you want to put in the list are numbers, an `array.array` is more efficient than a `list`: 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. 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 2-20 shows creating, saving and loading an array of 10 million floating-point random numbers:

Example 2-20. Creating, saving and loading a large array of floats.

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

- ❶ import the array type;
- ❷ create an array of double-precision floats (typecode 'd') from any iterable object, in this case a generator expression;
- ❸ inspect the last number in the array;
- ❹ save the array to a binary file;
- ❺ create an empty array of doubles;
- ❻ read 10 million numbers from the binary file;

- 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 show 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.



Another fast and 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`, but `pickle` handles almost all built-in types, including complex numbers, nested collections and even instances of user defined classes automatically — 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 endianness conversion
<code>s.clear()</code>	•		delete all items
<code>s.__contains__(e)</code>	•	•	<code>e in s</code>
<code>s.copy()</code>	•		shallow copy of the list
<code>s.__copy__()</code>		•	support for <code>copy.copy</code>
<code>s.count(e)</code>	•	•	count occurrences of an element
<code>s.__deepcopy__()</code>		•	optimized support for <code>copy.deepcopy</code>
<code>s.__delitem__(p)</code>	•	•	remove item at position <code>p</code>

	list	array	
<code>s.extend(it)</code>	•	•	append items from iterable <code>it</code>
<code>s.frombytes(b)</code>		•	append items from byte sequence interpreted as packed machine values
<code>s.fromfile(f, n)</code>		•	append <code>n</code> items from binary file <code>f</code> interpreted as packed machine values
<code>s.fromlist(l)</code>		•	append items from list; if one causes <code>TypeError</code> , none are appended
<code>s.__getitem__(p)</code>	•	•	<code>s[p]</code> — get item at position
<code>s.index(e)</code>	•	•	find position of first occurrence of <code>e</code>
<code>s.insert(p, e)</code>	•	•	insert element <code>e</code> before the item at position <code>p</code>
<code>s.itemsize</code>		•	length in bytes of each array item
<code>s.__iter__()</code>	•	•	get iterator
<code>s.__len__()</code>	•	•	<code>len(s)</code> — number of items
<code>s.__mul__(n)</code>	•	•	<code>s * n</code> — repeated concatenation
<code>s.__imul__(n)</code>	•	•	<code>s *= n</code> — in-place repeated concatenation
<code>s.__rmul__(n)</code>	•	•	<code>n * s</code> — reversed repeated concatenation ^a
<code>s.pop(«p»)</code>	•	•	remove and return item at position <code>p</code> (default: last)
<code>s.remove(e)</code>	•	•	remove first occurrence of element <code>e</code> by value
<code>s.reverse()</code>	•	•	reverse the order of the items in-place
<code>s.__reversed__()</code>	•		get iterator to scan items from last to first
<code>s.__setitem__(p, e)</code>	•	•	<code>s[p] = e</code> — put <code>e</code> in position <code>p</code> , overwriting existing item
<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 [Chapter 13](#).



As of Python 3.4, 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`](#)” on [page 46](#).

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 visit shortly in “NumPy and SciPy” on page 52). 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 — just like the C `cast` operator. `memoryview.cast` returns yet another `memoryview` object, always sharing the same memory.

See [Example 2-21](#) for an example of changing a single byte of an array of 16-bit integers.

Example 2-21. Changing the value of an array item by poking one of its bytes.

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

- ❶ Build `memoryview` from array of 5 short 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 typecode 'B' (unsigned char).
- ❹ Export elements of `memv_oct` as a list, 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 4-4](#)).

Meanwhile, if you are doing advanced numeric processing in arrays, then 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 applications⁹. 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 codebase from the [Netlib Repository](#). In other words, SciPy gives scientists the best of both worlds: an interactive prompt and high-level Python APIs, together with industrial-strength number crunching functions optimized in C and Fortran.

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

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

```
>>> import numpy ①
>>> a = numpy.arange(12) ②
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> type(a)
<class 'numpy.ndarray'>
>>> a.shape ③
(12,)
>>> a.shape = 3, 4 ④
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a[2] ⑤
array([ 8,  9, 10, 11])
```

9. Of course, if Python did not have an expressive and user-friendly syntax and semantics, and was not easy to extend in C, then the creators of NumPy and SciPy may not have invested in it. So it is fair to say these libraries are part of the reason why Python became successful among scientists in many fields; the rest of the credit goes to Guido van Rossum and the Python core team.

```

>>> a[2, 1] ❸
9
>>> a[:, 1] ❷
array([1, 5, 9])
>>> a.transpose() ❸
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;
- ❺ get row at index 2;
- ❻ get element at index 2, 1;
- ❼ get column at index 1;
- ❽ create a new array by transposing (swapping columns with rows).

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

```

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

```

- ❶ load 10 million floating point numbers from a text file;
- ❷ use sequence slicing notation to inspect the last 3 numbers;
- ❸ multiply every element in the `floats` array by `.5` and inspect last 3 elements again;
- ❹ import the high-resolution performance measurement timer (available since Python 3.3);

- ⑤ divide every element by 3; the elapsed time for 10 million floats is less than 40 milliseconds;
- ⑥ save the array in a `.npy` binary file;
- ⑦ 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;
- ⑧ inspect the last 3 elements after multiplying every element by 6.



Installing NumPy and SciPy from source is not a breeze. 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, installing them on Debian or Ubuntu is as easy as:

```
$ sudo apt-get install python-numpy python-sciipy
```

This was just an appetizer. NumPy and SciPy are formidable libraries, and are the foundation of other awesome tools such as the [Pandas](#) and [Blaze](#) data analysis libraries, which provide efficient array types that can hold non-numeric data as well as import/export functions compatible with many different formats like `.csv`, `.xls`, SQL dumps, HDF5 etc. 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.

Deque 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 LIFO behavior). But inserting and removing from the left of a list (the 0-index end) is costly because the entire list must be shifted.

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 maximum length and then, when it is full, it discards items from the opposite end when you append new ones.

```
>>> from collections import deque
>>> dq = deque(range(10), maxlen=10) ①
>>> 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 set the maximum number of items allowed in this instance of deque; this sets a read-only `maxlen` instance attribute;
- ❷ rotating with `n > 0` takes items from the right end and prepends them to the left; when `n < 0` items are taken from left and appended to the right;
- ❸ 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;
- ❹ adding three items to the right pushes out the leftmost -1, 1 and 2;
- ❺ 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 LIFO-queue in multi-threaded 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)

	list	deque
<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 [Chapter 13](#).

^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

Provides the synchronized (i.e. thread-safe) classes `Queue`, `LifoQueue` and `PriorityQueue`. These are used for safe communication between threads. All three classes 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 bounded `Queue`, very similar to `queue.Queue` but designed for inter-process communication. There is also has a specialized `multiprocessing.JoinableQueue` for easier task management.

`asyncio`

Newly added to Python 3.4, `asyncio` provides `Queue`, `LifoQueue`, `PriorityQueue` and `JoinableQueue` with APIs inspired by the classes in `queue` and `multiprocessing`, 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 4](#).

Chapter summary

Mastering the standard library sequence types is a pre-requisite 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.

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. Named tuples are not so new, but deserve more attention: like tuples, they have very little overhead per instance, yet provide convenient access to the fields by name and a handy `._asdict()` to export the record as an `OrderedDict`.

Sequence slicing is a favorite Python syntax feature, and it is even more powerful than many realize. Multi-dimensional 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 the Python Cookbook 3e. by David Beazley and Brian K. Jones has many recipes focusing on sequences, including recipe 1.11 — Naming a Slice — where I learned the trick of assigning slices to variables to improve readability, illustrated in our [Example 2-11](#).

The 2nd edition of the Python Cookbook was written for Python 2.4 but much of its code works with Python 3, and a lot of the recipes in chapters 5 and 6 deal with sequences. The Python Cookbook 2e 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 2nd edition solutions are no longer the best approach, I honestly think it is worthwhile to have both editions of the Python Cookbook on hand.

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

[PEP 3132 — Extended Iterable Unpacking](#) is the canonical source to read about the new use of `*extra` as a target in parallel assignments. If you'd like a glimpse of Python evolving, [Missing *-unpacking generalizations](#) is a bug tracker issue proposing even wider use of iterable unpacking notation. [PEP 448 — Additional Unpacking Generalizations](#) resulted from the discussions in that issue. As of this writing it seems likely the proposed changes will be merged to Python, perhaps in version 3.5.

Eli Bendersky has a short tutorial on `memoryview` at [Less copies in Python with the buffer protocol and memoryviews](#).

There are numerous books covering NumPy in the market, even some don't mention "NumPy" in the title. Wes McKinney's "Python for Data Analysis" (O'Reilly, 2012) is one such title.

Scientists love the combination of an interactive prompt with the power of NumPy and SciPy so much that they developed IPython, an incredibly powerful replacement for the Python console which also provides a GUI, integrated inline graph plotting, literate programming support (interleaving text with code), and rendering to PDF. Interactive, multimedia IPython sessions can even be shared over HTTP as IPython notebooks. See screenshots and video at [The IPython Notebook](#) page. IPython is so hot that in 2012 its core developers, most of whom are researchers at UC Berkeley, received a \$1.15 million dollar grant from the Sloan Foundation for enhancements to be implemented over the 2013-2014 period.

Chapter [8.3. collections — Container datatypes](#) of the Python Standard Library documentation has short examples and practical recipes using `deque` (and other collections).

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 Prof. Dijkstra explains with rigor and humor why the sequence 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 spell 2, 3, ..., 12 as `range(2, 13)`. By the way, the memo is a hand-written note, but it's beautiful and totally readable. Somebody should create a Dijkstra font. I'd buy it.

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 `frozenset` but is really tuple behaving as 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 the language became more flexible, more consistent and simpler at the same time. “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 from 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”. I can’t cite a reference to support the use of *flat sequence* in this specific context: as the category of Python sequence types that are not containers. In the Wikipedia this usage would be tagged “original research”. I prefer to call it “our term”, hoping you’ll find it useful and adopt it too.

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 comparable¹⁰.

```
>>> 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 on 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, while sorting Python also has to compare the keys, 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']
```

10. Python 2 lets you sort mixed lists, but you get the items grouped by type; within each group the items are sorted, but the order of the groups themselves is arbitrary and implementation-dependent, which means, “not dependable”.

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

Oracle, Google and the Timbot conspiracy

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

Timsort was first deployed in CPython, in 2002. Since 2009, Timsort is also used 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`.