Chapter 8. Lists and Dictionaries

Now that we’ve explored numbers and strings, this chapter moves on to give the full story on Python’s *list* and *dictionary* objects—collections of other objects, and the main workhorses in almost all Python scripts. As you’ll find, both are remarkably flexible: they can be changed in place, can grow and shrink on demand, and may contain and be nested in any other kind of object. By leveraging these built-in object types, you can create and process rich information structures in your scripts without having to define new object types of your own.

Lists

The first stop on this chapter’s tour is the Python *list*. Lists are Python’s most flexible ordered collection object type. Unlike strings, lists can contain any sort of object: numbers, strings, and even other lists. Also, unlike strings, lists may be changed in place by assignment to offsets and slices, list method calls, deletion statements, and more—they are *mutable* objects.

Python lists do the work of many of the collection data structures you might have to implement manually in lower-level languages such as C. Here is a quick look at their main properties. Python lists are:

Ordered collections of arbitrary objects

> From a functional view, lists are just places to collect other objects so you can treat them as groups. Lists also maintain a left-to-right positional ordering among the items they contain.

Accessed by offset

> Just as with strings, you can fetch a component object from a list by indexing the list on the object’s offset. Because items in lists are ordered by their positions, you can also do tasks such as slicing and concatenation.

Variable-length, heterogeneous, and arbitrarily nestable

Unlike strings, lists can grow and shrink in place (their lengths can vary), and they can contain any sort of object, not just one-character strings (they're heterogeneous). Because lists can both contain and be contained by other collection objects, they also support arbitrary nesting; you can create lists of lists of lists, and so on.

Of the category "mutable sequence"

In terms of our type category qualifiers, lists are mutable (i.e., can be changed in place) and can respond to all the *sequence* operations used with strings, such as indexing, slicing, and concatenation. In fact, sequence operations work the same on lists as they do on strings; the only difference is that sequence operations such as concatenation and slicing return new *lists* instead of new strings when applied to lists. Because lists are *mutable*, however, they also support other operations that strings don't, such as deletion, expansion, and index assignment operations, which change the lists in place.

Arrays of object references

Technically, Python lists contain zero or more *references* to other objects. Lists might remind you of arrays of pointers (addresses) if you have a background in some other languages, and fetching an item from a Python list is about as fast as indexing a C array. In fact, lists really *are* arrays inside the standard CPython interpreter, not linked structures. As we learned in Chapter 6, though, Python always follows a reference to an object whenever the reference is used, so your program deals only with objects. Whenever you assign an object to a data structure component or variable name, Python always stores a reference to that same object, not a copy of it (though the object stored may be a copy of another, if you requested one before the store).

As a preview and reference, Table 8-1 summarizes common and representative list object operations. It is fairly complete, but for the full story, consult the Python standard-library manual, or run a `help(list)` or `dir(list)` call interactively for a complete list of list methods—you can pass in a real list, or the word `list`, which is the name of the list data type. The set of methods here is especially prone to change, so be sure to cross-check in the future.

Table 8-1. Common list literals and operations

Operation	Interpretation
`L = []`	An empty list
`L = [123, 'abc', 1.23, {}]`	Four items: indexes 0..3
`L = ['Pat', 40.0, ['dev', 'mgr']]`	Nested sublists

`L = list('code')`
`L = list(range(-4, 4))`
	List of an iterable's items, list of successive integers

`L[i]`
`L[i][j]`
`L[i:j]`
`len(L)`
	Index, index of index, slice, length

`L1 + L2`
`L * 3`
	Concatenate, repeat

`L1 > L1, L1 == L2`	Comparisons: magnitude, equality

`3 in L`
`for x in L: print(x)`
	Membership, iteration

`L.append(4)`
`L.extend([5, 6, 7])`
`L.insert(i, X)`
	Methods: growing

`L.index(X)`
`L.count(X)`
	Methods: searching

`L.sort()`
`L.reverse()`
`L.copy()`
`L.clear()`
	Methods: sorting, reversing, copying, clearing

`L.pop(i)`
`L.remove(X)`
`del L[i]`
`del L[i:j]`
`L[i:j] = []`
	Methods, statements: shrinking

`L[i] = 3`
`L[i:j] = [4, 5, 6]`
	Index assignment, slice assignment

`L = [*x, 0, *y, *z]`	Iterable unpacking
`L = [x**2 for x in range(5)]`	List comprehensions and maps

Operation	Interpretation
`list(map(ord, 'python'))`	

When written down as a *literal* expression, a list is coded as a series of objects (really, expressions that return objects) in square brackets, separated by commas. For instance, the second row in Table 8-1 assigns the variable `L` to a four-item list. A *nested* list is coded as a nested square-bracketed series (row 3), and the *empty* list is just a square-bracket pair with nothing inside (row 1).[1]

Many of the operations in Table 8-1 should look familiar, as they are the same sequence operations we put to work on strings earlier—indexing, concatenation, iteration, and so on. Lists also respond to list-specific method calls (which provide utilities such as sorting, reversing, adding items to the end, etc.), as well as in-place change operations (deleting items, assignment to indexes and slices, and so forth). Again, lists have these tools for change operations because they are a mutable object type.

Lists in Action

Probably the best way to understand lists is to see them at work. Let's once again turn to some simple interpreter interactions to illustrate the operations in Table 8-1.

Basic List Operations

Because they are *sequences*, lists support many of the same operations as strings, which means we don't have to repeat all the operation details again here. In short, though, lists respond to the `+` and `*` operators much like strings—they mean concatenation and repetition here too, except that the result is a new list, not a string:

```
$ python3                                   # Launch c
>>> len([1, 2, 3])                          # Length
3
>>> [1, 2, 3] + [4, 5, 6]                   # Concaten
[1, 2, 3, 4, 5, 6]
>>> ['Py!'] * 4                             # Repetiti
['Py!', 'Py!', 'Py!', 'Py!']
```

Although the `+` operator works the same for lists and strings, it's important to know that it expects the *same* sort of sequence on both sides—otherwise, you get a type error when the code runs. For instance, you cannot concatenate a list and a string unless you first convert the list to a string (using tools such as `str` or formatting) or convert the string to a list (the `list` built-in function does the trick):

```
>>> str([1, 2]) + '34'                              # Same as
'[1, 2]34'
>>> [1, 2] + list('34')                             # Same as
[1, 2, '3', '4']
```

As suggested in prior chapters, lists also support *comparisons*, which automatically compare all parts from left to right until a result is known. In the following, for instance, a nested `3` is greater than a nested `2`, and the one-item list `[1]` at the end is considered less because it's shorter (though it broadly prefers the term horizontally challenged):

```
>>> L = [1, [2, 3], 4]
>>> (L == [1, [2, 3], 4]), (L > [1, [2, 2], 4]), (L > [
(True, True, True)
```

Indexing and Slicing

Because lists are sequences, indexing and slicing also work the same way for lists as they do for strings. For lists, though, the result of indexing is whatever type of object lives at the offset you specify (not a one-character string), and slicing a list always returns a new list (not a string):

```
>>> L = ['hack', 'Hack', 'HACK!']
>>> L[2]                                # Offsets start a
'HACK!'
>>> L[-2]                               # Negative: count
'Hack'
>>> L[1:]                               # Slicing fetches
['Hack', 'HACK!']
```

New here: because you can nest lists and other object types within lists, you will sometimes need to string together index operations to go *deeper* into a data structure. For example, one of the simplest ways to represent matrixes (multidimensional arrays) in Python is as lists with nested sublists. Here's a basic 3 × 3 two-dimensional list-based array—a reprise from Chapter 4:

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

With one index, you get an entire row (really, a nested sublist), and with two, you get an item within the row:

```
>>> matrix[1]
[4, 5, 6]
>>> matrix[1][1]
5
>>> matrix[2][0]
7
```

As demoed earlier, lists can naturally span multiple lines if you want them to because they are contained by a pair of brackets; watch for more on syntax like this in the next part of the book (and ignore the "…" if absent in your REPL):

```
>>> matrix = [[1, 2, 3],
...           [4, 5, 6],
...           [7, 8, 9]]
>>> matrix[1][1]
5
```

For more on matrixes, also watch for a dictionary-based alternative later in this chapter, which can be more efficient when matrixes are largely empty. We'll also continue this thread in Chapter 20 where we'll write additional matrix code, especially with list comprehensions. And for high-powered numeric work, the *NumPy* extension mentioned in Chapters 1, 4, and 5 provides other ways to handle matrixes.

Changing Lists in Place

Because lists are mutable, they support operations that change a list object *in place*. That is, list operations in this section and others all modify the list object directly—overwriting its former value—without requiring that you make a new copy, as you had to for strings. Because Python deals only in object references, this distinction between changing an object in place and creating a new object matters; as discussed in Chapter 6, if you change an object in place, you might impact more than one reference to it at the same time.

Index and slice assignments

First up in this category is a twist on the indexing and slicing we've already explored. When using a list, you can change its contents by *assigning* to either a particular item (*offset*) or an entire section (*slice*):

```
>>> L = ['code', 'Code', 'CODE!']
>>> L[1] = 'Hack'                   # Index assignmen
>>> L                               # Replaces item 1
['code', 'Hack', 'CODE!']

>>> L[0:2] = ['write', 'Python']    # Slice assignmen
>>> L                               # Replaces items
['write', 'Python', 'CODE!']
```

Both index and slice assignments are in-place changes—they modify the subject list directly, rather than generating a new list object for the result. *Index assignment* in Python works much as it does in most other languages: Python replaces the single object reference at the designated offset with a new one; the offset's reference is changed.

Slice assignment, the last operation in the preceding example, replaces an entire section of a list in a single step. Because it can be a bit complex, it is perhaps best thought of as a combination of two steps:

1. *Deletion*: The slice you specify to the left of the `=` is deleted.
2. *Insertion*: The new items contained in the iterable object to the right of the `=` are inserted into the list on the left, at the place where the old slice was deleted.[2]

This isn't what really happens, but it can help clarify why the number of items inserted doesn't have to match the number of items deleted. For instance, given a list `L` of two or more items, an assignment `L[1:2]=[4,5]` replaces one item with two—it's as though Python first deletes the one-item slice at `[1:2]` (from offset 1, up to but not including offset 2), then inserts both `4` and `5` where the deleted slice used to be. The net effect makes the list *larger*.

This also explains why the second slice assignment in the following is really an *insert*—Python replaces an empty slice at `[1:1]` with two items; and why the third is really a *deletion*—Python deletes the slice (the item at offset 1), and then inserts nothing:

```
>>> L = [1, 2, 3]
>>> L[1:2] = [4, 5]                       # Replacement/ins
>>> L
[1, 4, 5, 3]
>>> L[1:1] = [6, 7]                       # Insertion (repl
>>> L
[1, 6, 7, 4, 5, 3]
>>> L[1:2] = []                           # Deletion (inser
>>> L
[1, 7, 4, 5, 3]
```

In effect, slice assignment replaces an entire section, or "column," all at once —even if the column or its replacement is empty. Because the length of the sequence being assigned does not have to match the length of the slice being assigned to, slice assignment can be used to *replace* (by overwriting), *expand* (by inserting), or *shrink* (by deleting) the subject list. It's a powerful operation, but frankly, one that you may not see very often in practice. There are often more straightforward and mnemonic ways to replace, insert, and delete (including concatenation expressions, and the `insert`, `pop`, and `remove` list methods coming up soon), which Python programmers tend to prefer in practice.

On the other hand, this operation can be used as a sort of in-place concatenation at the front of the list—per the next section's method coverage, something the list's `extend` does more mnemonically but at list end:

```
>>> L = [1]
>>> L[:0] = [2, 3, 4]                # Insert all at :0, an e
>>> L
[2, 3, 4, 1]
>>> L[len(L):] = [5, 6, 7]     # Insert all at len(L):,
>>> L
[2, 3, 4, 1, 5, 6, 7]
>>> L.extend([8, 9, 10])       # Insert all at end, nam
>>> L
[2, 3, 4, 1, 5, 6, 7, 8, 9, 10]
```

List method calls

Like strings, Python list objects also support type-specific method calls, most of which change the subject list itself:

```
>>> L = ['write']
>>> L.append('Python')                  # Append: ac
>>> L
['write', 'Python']
>>> L.extend(['code', 'goodly'])        # Extend: ac
>>> L
['write', 'Python', 'code', 'goodly']
>>> L.sort()                            # Sort: orde
>>> L
['Python', 'code', 'goodly', 'write']
```

Methods were introduced in Chapter 7. In brief, they are functions (really, object attributes that reference functions) that are associated with and act upon particular subject objects (the objects through which they are called). Methods provide type-specific tools; the list methods presented here, for instance, are generally available only for lists.

Demoed in the preceding code, `append` is perhaps the most commonly used list method. It simply tacks a *single* item (really, an object reference) onto the end of the subject list *in place*, and the list expands to make room for the addition. Unlike concatenation, `append` expects you to pass in one object, not a list, and uses it literally. In fact, the effect of `L.append(X)` is similar to `L+[X]`, but the former changes `L` in place, while the latter makes a new list.[3]

By contrast, the `extend` method adds *multiple* items at the end of the list, again in place. Technically, `extend` always iterates through and adds each item in the passed *iterable* object, whereas `append` simply adds a single item as is without iterating—a distinction that will be more meaningful in Chapter 14. For now, it's enough to know that `extend` adds many items, and `append` adds one. Also ahead: Chapter 11 covers the `+=` statement, which does in-place assignment for lists too, and is yet another way to add many items that redundantly mirrors `extend`.

The `sort` method *orders* the list's items but merits a section of its own.

Sorting lists

As we've just witnessed, `sort` orders a list in place. It's a common tool that uses Python standard comparison tests (string comparisons in most examples here, but other objects work in sorts too), and by default sorts in ascending order. You can modify sort behavior by passing in *keyword arguments*—a special *`name=value`* syntax in function calls that we've used in earlier chapters, gives values by name, and is often used for configuration options.

In sorts, the `reverse` argument allows sorts to be made in descending instead of ascending order, and the `key` argument gives a one-argument function that returns the value to be used in sorting—the string object's standard `lower` case converter in the following (though its newer `casefold` may handle some types of Unicode text better):

```
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort()                                 # Sort with
>>> L
['ABD', 'aBe', 'abc']

>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower)                    # Normalize
>>> L
['abc', 'ABD', 'aBe']

>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower, reverse=True)      # And chang
>>> L
['aBe', 'ABD', 'abc']
```

The sort `key` argument can also be useful when sorting lists of dictionaries, to select a sort value by indexing each dictionary on a field along the way. We'll study dictionaries later in this chapter, and you'll learn more about keyword function arguments in Part IV.

Two notes of caution here: first, sorts won't work by default with *mixed types*. In Python, magnitude comparison of mixed types is an error, as it's generally ambiguous. Because sorting uses these comparisons internally, though, sorting mixed-type lists fails by proxy. To work around this limitation, use the `key` argument to code value transformations during the sort. The following simply converts all items to strings with the `str` built-in, but `key` can be an arbitrary function of your making (and is often coded inline with the `lambda` expression you'll meet later in this book):

```
>>> L = [1, 'hack', 2]                # Mixed-type sorts fc
>>> L.sort()
TypeError: '<' not supported between instances of 'str'

>>> L.sort(key=str)
>>> L                                 # Enable mixed-type s
[1, 2, 'hack']
```

Second, beware that `append`, `extend`, and `sort` change the associated list object *in place*, but don't return the modified list as a result (technically, they return the `None` placeholder object introduced in Chapter 4). If you run `L=L.append(X)`, you won't get the modified value of `L`; in fact, you'll lose the reference to the list altogether! When you use methods like these, objects are changed as a side effect, so there's no reason to reassign.

Partly because of such constraints, sorting is also available in Python as the `sorted` built-in *function*, which sorts *any* collection (not just lists) and returns a *new* list for the result (instead of changing the collection in place):

```
>>> L = ['abc', 'ABD', 'aBe']
>>> sorted(L)                                         # Sc
['ABD', 'aBe', 'abc']
>>> L                                                 # L
['abc', 'ABD', 'aBe']

>>> sorted(L, key=str.lower, reverse=True)            # Sc
```

```
['aBe', 'ABD', 'abc']

>>> L = ['abc', 'ABD', 'aBe']
>>> sorted([x.lower() for x in L], reverse=True)    # Pr
['abe', 'abd', 'abc']
```

Notice the last example here—we can convert to lowercase prior to the sort with a list comprehension (introduced in Chapter 4 and expanded shortly), but the result does not contain the *original* list's values as it does with the `key` argument. The latter is applied temporarily during the sort, instead of changing the values to be sorted altogether. As we move along, you'll see roles in which the `sorted` built-in can sometimes be more useful than the `sort` method.

More List Methods

Like strings, lists have other methods that perform other specialized operations. For instance, `reverse` reverses the list in place, and `pop` deletes one item at the end (by default). Just like sorting, there is also a `reversed` built-in function that does not change the list in place. Confusingly, though, `reversed` does not return a new list like `sorted`; it instead returns an iterable object that produces results on demand, and must be wrapped in a `list` call to collect its results if a real list is needed (e.g., for indexing, or display at the REPL here):

```
>>> L = [1, 2, 3, 4, 5]
>>> L
[1, 2, 3, 4, 5]
>>> L.pop()                            # Delete and retur
5
>>> L
[1, 2, 3, 4]
>>> L.reverse()                        # In-place revers
>>> L
[4, 3, 2, 1]
>>> list(reversed(L))                  # Reversal built-i
[1, 2, 3, 4]
>>> L                                  # L was unchanged
[4, 3, 2, 1]
```

In some types of programs, the list `pop` method is used in conjunction with `append` to implement a quick last-in-first-out (LIFO) *stack* structure. The end of the list serves as the “top” of the stack:

```
>>> L = []
>>> L.append(1)                         # Push onto stack
>>> L.append(2)
>>> L
[1, 2]
>>> L.pop()                             # Pop off stack
2
>>> L
[1]
```

The `pop` method also accepts an optional offset of the item to be deleted and returned (the default is the last item at offset −1). Other list methods remove an item by value (`remove`), insert an item at an offset (`insert`), count the number of occurrences (`count`), and search for an item’s offset (`index` —a search for the *index of* an item, not to be confused with indexing itself, despite the name!):

```
>>> L = ['hack', 'Py', 'code']
>>> L.index('Py')                       # Index _of_ an ob
1
>>> L.insert(1, 'more')                 # Insert at offset
>>> L
['hack', 'more', 'Py', 'code']
>>> L.remove('code')                    # Delete by value
>>> L
['hack', 'more', 'Py']
>>> L.pop(1)                            # Delete by offset
'more'
>>> L
['hack', 'Py']
>>> L.count('Py')                       # Number of occurr
1
```

Note that unlike other list methods, `count` and `index` do not change the list itself, but return information about its content. Run a `help(list)` or dir(list) in your REPL, see Python reference resources, or experiment with these calls interactively on your own to learn more about list methods.

Iteration, Comprehensions, and Unpacking

Lists also respond to other sequence operations we used on strings in the prior chapter, including *iteration* tools, and are regularly used in conjunction with the `range` built-in previewed in Chapter 4:

```
>>> 3 in [1, 2, 3]                                        # Membersh
True
>>> for x in [1, 2, 3]:
...     print(x, end=' ')                                 # Iteratic
...
1 2 3
>>> list(range(5))                                        # Counter
[0, 1, 2, 3, 4]
```

We will talk more formally about `for` iteration and the `range` built-in in Chapter 13, because they are related to statement syntax. Per Chapter 4 previews, though, `for` loops step through items in any sequence (or other iterable) from left to right, executing one or more statements for each item; and `range` makes a series of integers, for use in a variety of roles, including counter loops (and coerced by `list` to surrender its values for display here).

List comprehensions and maps

The last items in Table 8-1, list comprehensions and `map` calls, were also introduced in Chapter 4 and are covered in full in Chapters 14 and 20. Their basic usage is straightforward, though—list *comprehensions* are close kin to `for` loops, and build a new list by applying an expression to each item in a sequence (really, any iterable):

```
>>> res = [x * 4 for x in 'code']                         # List com
>>> res
['cccc', 'oooo', 'dddd', 'eeee']
```

As a preview, this expression is equivalent to a `for` loop that builds up a list of results manually with the `append` we met earlier, but as you'll learn in later chapters, list comprehensions are simpler to code and may run faster:

```
>>> res = []                                 # Equivale
>>> for x in 'code':                         # Append e
...     res.append(x * 4)
...
>>> res
['cccc', 'oooo', 'dddd', 'eeee']
```

Comprehension also support extensions like `if` filters and nested `for` loops, and the `map` built-in function does similar work, but applies a *function* instead of an expression to items in a sequence (or other iterable) and collects all the results in a new list—if we wrap it in `list` to force it to produce all its results:

```
>>> [x * 4 for x in 'program' if x >= 'p']   # Filter i
['pppp', 'rrrr', 'rrrr']

>>> [x + y for x in 'py' for y in '312']     # Nested l
['p3', 'p1', 'p2', 'y3', 'y1', 'y2']

>>> list(map(abs, [-1, -2, 0, 1, 2]))        # Map a fu
[1, 2, 0, 1, 2]
```

While these are powerful tools, their loop equivalents also make them optional. Stay tuned for the related dictionary comprehension later in this chapter, and much more on comprehensions and maps later in this book.

List-literal unpacking

As of Python 3.5, list literals also support a `*` syntax that *unpacks* the contents of any iterable (including sequences like lists and strings) at the top level of the list being created. The effect flattens the starred item in the new list:

```
>>> L = ['code', 'hack']
>>> [L, 2, 3, L]
[['code', 'hack'], 2, 3, ['code', 'hack']]   # Normal c

>>> [*L, 2, 3, *L]                           # Iterable
```

```
['code', 'hack', 2, 3, 'code', 'hack']

>>> S = 'code'                              # Any sequ
>>> [*'Py', *S, *range(3)]
['P', 'y', 'c', 'o', 'd', 'e', 0, 1, 2]
```

Importantly, and as is so often true in Python today, `*` unpacking is never required. The following simple *concatenation*, for example, has the same effect as the preceding's first `*` unpacking—which is redundant in this case:

```
>>> L + [2, 3] + L                          # What * u
['code', 'hack', 2, 3, 'code', 'hack']
```

For nonlist iterables, unpacking without `*` requires *conversions* because `+` expects lists on both sides, but it's not much extra work, especially given the rarity of such code:

```
>>> list('Py') + list(S) + list(range(3))    # What * u
['P', 'y', 'c', 'o', 'd', 'e', 0, 1, 2]
```

Loops can achieve the same goals as unpacking too; they may be more verbose, but are also more general purpose:

```
>>> M = []
>>> for x in ('Py', S, range(3)): M.extend(x)        #
...
>>> M
['P', 'y', 'c', 'o', 'd', 'e', 0, 1, 2]
```

Notice two things about this example. First, it codes a *tuple* of three items as the sequence that the `for` loop steps through (the tuple's outer parentheses are optional but explicit). Second, the `for` loop in this code avoids a line by moving its body up to its *header*, using a syntax rule we'll define formally in the next part of this book (tl;dr: this works only for simple statements, like calls to `print` and `extend`).

Because we're not quite ready for the full iteration story, we'll postpone further details for now, but watch for similar unpacking expressions for dictionaries later in this chapter. Here, though, you should already be able to weigh for yourself whether the alternatives to `*` unpacking were sufficiently subpar to warrant the convolution.

Other List Operations

Because lists are mutable, you can also use the `del` statement to delete an item or section in place:

```
>>> L = ['hack', 'more', 'Py', 'code']
>>> del L[0]                                    # Delete one i
>>> L
['more', 'Py', 'code']
>>> del L[1:]                                   # Delete an er
>>> L                                           # Same as L[1:
['more']
```

As covered earlier, because *slice assignment* is a deletion plus an insertion, you can also delete a section of a list by assigning an empty list to a slice (`L[i:j]=[]`); Python deletes the slice named on the left, and then inserts nothing. Assigning an empty list to an index, on the other hand, just stores a reference to the empty list object in the specified slot, rather than deleting an item:

```
>>> L = ['hack', 'more', 'Py', 'code']
>>> L[0:1] = []
>>> L
['more', 'Py', 'code']
>>> L[1:] = []
>>> L
['more']
>>> L[0] = []
>>> L
[[]]
```

Although all the operations just discussed are typical, there may be additional list methods and operations not illustrated here. The method toolbox may also change over time, and in fact has: the newer `L.copy()` method makes a top-level copy of the list, much like `L[:]` and `list(L)`, but is symmetric with `copy` methods in sets and dictionaries. For a comprehensive and up-to-date list of type tools, you should always consult Python's manuals.

And because it's such a common hurdle, this book is compelled to remind you once again that all the in-place change operations discussed here work only for *mutable* objects: they won't work on strings (or tuples, coming up in Chapter 9), no matter how hard you try. Mutability is an inherent yes/no property of each object type—including the subject of this chapter's next section.

Dictionaries

Along with lists, *dictionaries* are one of the most flexible built-in object types in Python. If you think of lists as order-based collections of objects, you can think of dictionaries as key-based collections; the chief distinction is that in dictionaries, items are stored and fetched by *key*, instead of by positional offset. While lists can serve roles similar to arrays in other languages, dictionaries can take the place of records, search tables, and any other sort of aggregation where item names are more meaningful than item positions.

For example, dictionaries can replace many of the searching algorithms and data structures you might have to implement manually in lower-level languages—as a highly optimized built-in tool, indexing a dictionary is a very fast search operation. Dictionaries also sometimes do the work of records and symbol tables used in other languages, can be used to represent sparse (mostly empty) data structures, and much more. As a rundown of their main properties, Python dictionaries are:

Accessed by key, not offset position

> Dictionaries are sometimes called *associative arrays* or *hashes* (especially by users of other scripting languages). They associate a set of values with corresponding keys, so you can fetch an item out of a dictionary using the key under which you originally stored it. You use the same indexing operation to get components in a dictionary as you do in a list, but the index takes the form of a key, not a relative offset.

Insertion-ordered collections of arbitrary objects

> Unlike in a list, keys in a dictionary are ordered only by the order in which they are *inserted*. This is not the same as the positional ordering

of sequences like lists: items added to dictionaries later always show up at the end of the keys list (even if they appeared earlier in the past), and there is no way to insert a key into the middle of the keys list. Moreover, this ordering is new as of Python 3.7, before which keys' left-to-right order was pseudo random. Under either ordering regime, keys provide the symbolic (not physical) locations of items in a dictionary.

Variable-length, heterogeneous, and arbitrarily nestable

Like lists, dictionaries can grow and shrink in place (without new copies being made), they can contain objects of any type, and they support nesting to any depth (they can be freely mixed with lists, other dictionaries, and so on). Each key can have just one associated value, but that value can be a *collection* of multiple objects if needed, and a given value can be stored under any number of keys.

Of the category "mutable mapping"

You can change dictionaries in place by assigning to indexes (they are *mutable*), but they don't support the sequence operations that work on strings and lists. Because dictionaries are key-based collections, operations that depend on a fixed positional order (e.g., concatenation, slicing) don't make sense. Instead, dictionaries are the only built-in, core-type representatives of the *mapping* category—objects that map keys to values. Other mappings in Python are created by imported modules, not language syntax.

Tables of object references (hash tables)

If lists are arrays of object references that support access by position, dictionaries are tables of object references that support access by key. Internally, dictionaries are implemented as hash tables (data structures that support very fast retrieval), which start small and grow on demand. Moreover, Python employs optimized hashing algorithms to find keys, so retrieval is quick. Like lists, though, dictionaries store object *references* (not copies, unless you make them explicitly before storing).

For reference and preview, Table 8-2 summarizes some of the most common and representative dictionary operations and is relatively complete at this writing. As usual, though, you should consult Python's library manual or run a `dir(dict)` or `help(dict)` call for a complete list (`dict` is the built-in name of the dictionary type).

Table 8-2. Common dictionary literals and operations

Operation	Interpretation
`D = {}`	Empty dictionary
`D = {'name': 'Pat', 'age': 40.0}`	Two-item dictionary
`E = {'cto': {'name': 'Sue', 'age': 40}}`	Nesting

`D = dict(name='Bob', age=40)`
`D = dict([('name', 'Bob'), ('age', 40)])`
`D = dict(zip(keyslist, valueslist))`
`D = dict.fromkeys(['name', 'age'])`
	Alternative construction techniques: keywords, key/value pairs, zipped key/value lists, key lists

`D['name']`
`E['cto']['age']`
	Indexing by key, nested indexes

`'age' in D`
`for key in D: print(D[key])`
	Key membership and iteration

`D.keys()`
`D.values()`
`D.items()`
`D.copy()`
`D.clear()`
`D.update(D2)`
`D.get(key, default?)`
`D.pop(key, default?)`
`D.setdefault(key, default?)`
`D.popitem()`
	Methods: all keys,
	all values,
	all key+value tuples,
	copy (top-level),
	clear (remove all items),
	merge by keys,
	fetch by key, if absent default (or None),
	remove by key, if absent default (or error)
	fetch by key, if absent set default (or None),
	remove/return any (key, value) pair; etc.

`len(D)`	Length: number of stored entries
`D[key] = 62`	Adding keys, changing key values
`del D[key]`	Deleting entries by key
`D1 == D2`	Comparisons: equality (only)

`list(D.keys())`
`D1.keys() & D2.keys()`
	Dictionary views

Operation	Interpretation
`D = {**x, 'a': 1, **y, **z}`	Dictionary unpacking
`D = {x: x*2 for x in range(10)}`	Dictionary comprehensions
`D | E`	Dictionary merge expression: copy + update

Per Table 8-2, when coded as a *literal* expression, a dictionary is written as a series of *`key:value`* pairs separated by commas, and enclosed in curly braces.[4] An empty dictionary is an empty set of curly braces, and you can nest dictionaries by simply coding one as a value inside another dictionary, or within a list or tuple literal. There's a lot more to dictionaries than their literals, though, as the next section's tutorial will begin to reveal.

Dictionaries in Action

As Table 8-2 summarizes, dictionaries are indexed by key, and nested dictionary entries are referenced by a series of indexes (keys in square brackets). When Python creates a dictionary, it stores its items in a way that associates values with keys; to fetch a value back, you supply the key with which it is associated, not its relative position. Let's go back to the interactive prompt to see what this all looks like in code.

Basic Dictionary Operations

In normal operation, you create dictionaries with literals and store and access items by key with indexing:

```
$ python3
>>> D = {'hack': 1, 'Py': 2, 'code': 3}        # Make a
>>> D['Py']                                    # Fetch a
2
>>> D                                          # Inserti
{'hack': 1, 'Py': 2, 'code': 3}
```

Here, the dictionary is assigned to the variable `D`; the value of the key `'Py'` is the integer `2`, and so on. We use the same square bracket syntax to index dictionaries by key as we did to index lists by offset, but here it means access by key, not by position. Notice the end of this example: unlike the randomly ordered sets we studied in Chapter 5, dictionary keys retain their insertion order today, but we need to cover some more basics before formalizing this.

The built-in `len` function works on dictionaries, too—it returns the number of values stored in the dictionary or, equivalently, the number of its keys. The dictionary `in` membership operator allows you to test for key existence, and the `keys` method returns all the keys in the dictionary. The latter of these can be useful for processing dictionaries in full, by fetching corresponding values in loops. Because the `keys` result can be treated like a normal list, it can also be sorted if order matters and the automatic insertion order doesn't do the job (more on sorting and dictionaries later):

```
>>> len(D)                                           # Number
3
>>> 'code' in D                                      # Key men
True
>>> list(D.keys())                                   # Create
['hack', 'Py', 'code']
```

Notice the second step in this listing. As noted, the `in` membership test used for strings and lists also works on dictionaries, where it checks whether a key is stored. Technically, this works because dictionaries define a protocol run by `in` to lookup a key quickly. Other objects provide `in` protocols that reflect their common uses; files, for example, use iterators that read line by line. This will matter later in this book, when we reach iterators and classes.

Also note the syntax of the last step in this listing. It uses `list` for similar reasons—the dictionary `keys` method returns an `iterable` "view" object that produces results on demand, instead of a physical list. The `list` call forces it to serve up all its values at once for display at the interactive REPL, though this call isn't required and may even be subpar in some other contexts. More on this, as well as other dictionary basics like comparisons, later in this chapter.

Changing Dictionaries in Place

Let’s continue with our interactive session. Dictionaries, like lists, are *mutable*, so you can change, expand, and shrink them in place without making new dictionaries: simply assign a value to a key to change or create an entry. The `del` statement works here, too; it deletes the entry associated with the key specified as an index. Notice also the nesting of a list inside a dictionary in this example (the value of the key `'Py'`); all collection data types in Python can nest inside each other arbitrarily, and can be changed independently of their containing objects:

```
>>> D
{'hack': 1, 'Py': 2, 'code': 3}

>>> D['Py'] = ['app', 'dev']                           # Cha
>>> D
{'hack': 1, 'Py': ['app', 'dev'], 'code': 3}

>>> del D['code']                                      # Del
>>> D
{'hack': 1, 'Py': ['app', 'dev']}

>>> D['years'] = 32                                    # Add
>>> D
{ {'hack': 1, 'Py': ['app', 'dev'], 'years': 32}

>>> D['Py'][0] = 'program'                             # Cha
>>> D
{'hack': 1, 'Py': ['program', 'dev'], 'years': 32}
```

Like lists, assigning to an existing index in a dictionary changes its associated value. Unlike lists, however, whenever you assign a *new* dictionary key (one that isn’t already present) you create a new entry in the dictionary, as was done in the previous example for the key `'years'`. This doesn’t work for lists because you can only assign to existing list offsets—Python considers an offset beyond the end of a list out of bounds and raises an error. As you learned earlier, to expand a list, you need to use tools such as the `append` method or slice assignment instead.

More Dictionary Methods

Dictionary methods provide a variety of type-specific tools. For instance, the dictionary `values` and `items` methods return all of the dictionary's values and `(key,value)` pair tuples, respectively; along with `keys`, these are useful in loops that need to step through dictionary entries one by one (we'll start coding such loops later in this chapter). As with `keys`, these two methods also return *iterable* objects, and wrapping them in a `list` call collects their values all at once for display:

```
>>> D = {'program': 1, 'script': 2, 'app': 3}

>>> list(D.keys())                                         # Al
['program', 'script', 'app']
>>> list(D.values())                                       # Al
[1, 2, 3]
>>> list(D.items())                                        # Al
[('program', 1), ('script', 2), ('app', 3)]
```

In realistic programs that gather data as they run, you often won't be able to predict what will be in a dictionary before the program is launched, much less when it's coded. Fetching a nonexistent key is normally an error, but the `get` method returns a default value— `None`, or a passed-in default—if the key doesn't exist. It's an easy way to fill in a default for a key that isn't present and avoid a missing-key error when your program can't anticipate contents ahead of time:

```
>>> D.get('script')                                # A key that
2
>>> print(D.get('code'))                           # A key that
None
>>> D.get('code', 4)
4
```

The `update` method provides something similar to concatenation for dictionaries (though it works in the realm of keys and values, not just values). It *merges* the keys and values of one dictionary into another, both adding new entries for new keys, and blindly overwriting values of the same key if there's a clash:

```
>>> D
{'program': 1, 'script': 2, 'app': 3}
>>> D2 = {'code':4, 'hack':5, 'app': 6}     # New keys c
>>> D.update(D2)
>>> D
{'program': 1, 'script': 2, 'app': 6, 'code': 4, 'hack'
```

Finally, the dictionary `pop` method deletes a key from a dictionary and returns the value it had. It's similar to the list `pop` method, but it takes a key instead of an optional position:

```
>>> D.pop('app')                            # Pop a dict
6
>>> D.pop('hack')                           # Delete key
5
>>> D
{'program': 1, 'script': 2, 'code': 4}

>>> L = ['aa', 'bb', 'cc', 'dd']            # Pop a list
>>> L.pop()                                 # Delete and
'dd'
>>> L.pop(1)                                # Delete fro
'bb'
>>> L
['aa', 'cc']
```

Dictionaries also provide a `copy` method that, as you might guess, makes a copy; we'll revisit this in Chapter 9, as it's a way to avoid the potential side effects of shared references to the same dictionary. In fact, dictionaries come with more methods than the common ones demoed here, and over time may gain others beyond those listed in Table 8-2; again, see the Python library manual, `dir` and `help`, or other reference resources for a comprehensive list.

Other Dictionary Makers

Because dictionaries are so useful, multiple ways to build them have emerged over time. The following summarizes the most common alternatives; its last two calls to the `dict` constructor (really, type name) have the same effect as the literal and key-assignment forms listed above them:

```
{'name': 'Pat', 'age': 40}             # 1) Traditionc

D = {}                                 # 2) Assign by
D['name'] = 'Pat'
D['age']  = 40

dict(name='Pat', age=40)               # 3) dict keywc

dict([('name', 'Pat'), ('age', 40)])   # 4) dict key/v
```

All four of these forms create the same two-key dictionary, but they are useful in differing circumstances:

- The first is handy if you can spell out the entire dictionary ahead of time.
- The second is of use if you need to create the dictionary one field at a time on the fly.
- The third involves less typing than the first, but it requires all keys to be strings.
- The last is useful if you need to build up keys and values as sequences at runtime.

We met *`name=value`* keyword arguments earlier when sorting lists; the `dict` form in this summary that uses them is newer than literals and common in Python code, because it has less syntax (and hence less room for mistakes). The last form in the summary is also commonly used in conjunction with the `zip` function, to combine separate lists of keys and values obtained dynamically at runtime (parsed out of a data file's columns, for instance):

```
dict(zip(keyslist, valueslist))       # Zipped key/val
```

There is more on zipping dictionary keys in the next section. Provided all the key's values are the same initially, you can also create a dictionary with the following special form—simply pass in a list of keys and an initial value for all of the values (the default is `None`). Notice this is run from the `dict` type name, not an actual dictionary:

```
>>> dict.fromkeys(['a', 'b'], 0)
```

```
{'a': 0, 'b': 0}
```

Although you could get by with just literals and key assignments at this point in your Python career, you'll probably find uses for all of these dictionary-creation forms as you start applying them in realistic Python programs.

Dictionary-literal unpacking

As of Python 3.5, dictionary *literals* also support a special `**` syntax that *unpacks* the contents of another dictionary in its top level, similar to the `*` in list literals we met earlier. Any number of `**` can appear in a literal to unpack any number of dictionaries, and the rightmost's value wins when keys collide:

```
>>> D = dict(a=4, c=3)
>>> {'a': 1, 'b': 2, **D}                            # Dictior
{'a': 4, 'b': 2, 'c': 3}

>>> dict(a=1, b=2) | D                               # Same, t
{'a': 4, 'b': 2, 'c': 3}
```

As shown, the effect of `**` is similar to both the `update` method we met earlier and the dictionary `|` union covered ahead and might avoid a follow-up `update` call in some contexts. It's also related to the extended-unpacking assignment statement you'll meet in Chapter 11.

The `**` also works in `dict`, but simply because `**` unpacks keyword arguments in *any* function call (as usual in Python, this relies on larger concepts we'll reach later in this book). Unlike in `{}` literals though, `**` in `dict` fails if any key appears twice:

```
>>> dict(a=1, **{'b': 2}, **dict(c=3))               # Works i
{'a': 1, 'b': 2, 'c': 3}                             # As Long

>>> dict(a=1, b=2, **dict(a=4, c=3))
TypeError: dict() got multiple values for keyword argum
```

The examples so far demo the many basic ways to create dictionaries, but there is yet another that's even more powerful, as the next section will explain.

Dictionary Comprehensions

Like the set and list comprehensions we met in previous coverage, dictionary comprehensions run an implied loop, collecting the key/value results of expressions on each iteration and using them to fill out a new dictionary. A loop variable allows the comprehension to use loop iteration values along the way. The net effect lets us create new dictionaries with small bits of code that are simpler than the full-blown statements we'll study later in this book.

Abstractly, dictionary comprehensions map to `for` loops as follows, where both `k` and `v` can use loop variable `x` :

```
{k: v for x in iterable}          # Dictionary comprehens

new = {}                          # Equivalent loop code
for x in iterable:
    new[k] = v
```

To illustrate, a standard way to initialize a dictionary dynamically is to combine its keys and values with `zip` , and pass the result to the `dict` call, per the last section. The `zip` built-in function is the hook that allows us to construct a dictionary from key and value lists this way—if you cannot predict the set of keys and values in your code, you may be able to build them up as lists and zip them together. We'll study `zip` in detail in the next part of this book; it's an iterable, so we must wrap it in a `list` call to show its results there, but its basic usage is otherwise straightforward:

```
>>> list(zip(['a', 'b', 'c'], [1, 2, 3]))        # Zip
[('a', 1), ('b', 2), ('c', 3)]

>>> D = dict(zip(['a', 'b', 'c'], [1, 2, 3]))    # Make
>>> D
{'a': 1, 'b': 2, 'c': 3}
```

You can achieve the same effect, though, with a dictionary *comprehension* expression. The following uses tuple assignment to unpack items into variables (per its Chapter 4 debut), as it scans the list of zipped pairs from left to right. The net effect builds a new dictionary with a key/value pair for every such pair in the `zip` result (the Python code reads almost the same as this natural-language description, but with a bit more formality):

```
>>> D = {k: v for (k, v) in zip(['a', 'b', 'c'], [1, 2,
>>> D
{'a': 1, 'b': 2, 'c': 3}
```

Comprehensions are longer to code in this case, but they're also more general than this example implies—we can use them to map a single stream of values to dictionaries and apply them to any kind of sequence (or other iterable), and collected keys can be computed with expressions just like collected values:

```
>>> D = {x: x ** 2 for x in [1, 2, 3, 4]}          # Or:
>>> D
{1: 1, 2: 4, 3: 9, 4: 16}

>>> D = {c: c * 4 for c in 'HACK'}                 # Loop
>>> D
{'H': 'HHHH', 'A': 'AAAA', 'C': 'CCCC', 'K': 'KKKK'}

>>> D = {c.lower(): (c + '!') for c in ['HACK', 'PY', '
>>> D
{'hack': 'HACK!', 'py': 'PY!', 'code': 'CODE!'}
```

Dictionary comprehensions are also useful for initializing dictionaries from keys lists, in much the same way as the `fromkeys` method we met at the end of the preceding section:

```
>>> D = dict.fromkeys(['a', 'b', 'c'], 0)          # Init
>>> D
{'a': 0, 'b': 0, 'c': 0}

>>> D = {k: 0 for k in ['a', 'b', 'c']}            # Same
>>> D
{'a': 0, 'b': 0, 'c': 0}

>>> D = dict.fromkeys('code')                      # Othe
>>> D
{'c': None, 'o': None, 'd': None, 'e': None}

>>> D = {k: None for k in 'code'}
>>> D
{'c': None, 'o': None, 'd': None, 'e': None}
```

Like its list and set relatives, dictionary comprehensions support additional syntax not shown here, including `if` clauses to filter values out of results, and nested `for` loops. Unfortunately, to truly understand dictionary comprehensions, you need to also know more about iteration statements and concepts in Python, and this book hasn't yet disclosed enough information to tell that story well. You'll learn much more about all flavors of comprehensions—list, set, dictionary, and generator—in Chapters 14 and 20, so we'll defer further details until later.

Key Insertion Ordering

By now, you've probably noticed that dictionaries remember their keys' order—no matter what sort of syntax we use to make them. As noted both earlier here and in Chapter 4, as of Python 3.7 (and CPython 3.6), dictionaries have shed their former random key order and adopted *insertion order* instead: keys are now ordered left to right from oldest to newest additions. This order is maintained for literals, `dict` calls, comprehensions, and new keys added on the fly.

Among other things, insertion order makes dictionary outputs arguably more coherent and readable and may avoid key sorts in some programs. This isn't the same as a list (e.g., there's no way to add a key in the "middle") and doesn't make dictionaries sequences. The effect, however, is close. Here's a brief illustration:

```
>>> D = dict(a=1, b=2)              # Literal keys stor
>>> D['c'] = 3                      # New keys always a
>>> D
{'a': 1, 'b': 2, 'c': 3}
>>> D.pop('b')                      # Method pop remove
2
>>> D                               # Other keys still
{'a': 1, 'c': 3}
>>> D['b'] = 2                      # Earlier key goes
>>> D
{'a': 1, 'c': 3, 'b': 2}
>>> D['c'] = 0                      # Changing values a
>>> D['d'] = 1                      # And new arrivals
>>> D
{'a': 1, 'c': 0, 'b': 2, 'd': 1}
```

If you stare at this long enough, you'll probably notice that the insertion order of dictionaries is much like the LIFO *stack* ordering we coded with list `append` and `pop` methods earlier. In fact, the dictionary `popitem` is defined to return the key/value pair (really, tuple) for the key added most recently (i.e., at the end):

```
>>> D.popitem()
('d', 1)
>>> D.popitem()
('b', 2)
>>> D
{'a': 1, 'c': 0}
```

While useful in some contexts, keep in mind that you may still need to *sort* dictionary keys manually, using techniques we'll study ahead. Ordering filenames for display, for example, will often warrant string-value sorts instead of a program's internal insertion order.

Dictionary "Union" Operator

Last up in the dictionary toolbox is a bit of an oddball: as of Python 3.9, dictionaries have sprouted a "union" operation that's kicked off with the `|` operator—the same syntax used for union in the set objects you met in Chapter 5.

Per the quotes, though, this operation isn't really mathematical set union at all. It's a positionally sensitive, key-based *merge* of dictionaries, that works the same as the dictionary `update` method but *returns* its result as a new dictionary instead of updating a subject in place. Hence, dictionary `|` is like the combination of running the dictionary's `copy` and `update` methods in turn, and really just a minor convenience for narrow roles. Here it is at work:

```
>>> D = dict(a=1, b=2)
>>> D
{'a': 1, 'b': 2}

>>> D | {'b': 3, 'c': 4}                # Update, return,
{'a': 1, 'b': 3, 'c': 4}

>>> D | {'b': 3, 'c': 4} | dict(a=5, d=6)
{'a': 5, 'b': 3, 'c': 4, 'd': 6}
```

You get roughly the same mileage with the dictionary's longstanding `update` method, though it requires a manual `copy` to avoid in-place changes in programs that don't want them (as a spoiler, the `|=` in-place assignment form of dictionary union that you'll meet later in this book is *identical* to calling `update` with a single argument sans `copy`):

```
>>> D
{'a': 1, 'b': 2}
>>> C = D.copy()
>>> C.update({'b': 3, 'c': 4})        # "Union" sans the
>>> C.update(dict(a=5, d=6))
>>> C
{'a': 5, 'b': 3, 'c': 4, 'd': 6}
```

The Python 3.5 dictionary-literal *unpacking* we explored earlier can have the same effect too, though, at least when used with gnarly embedded literals as in the following, perhaps less readably:

```
>>> D
{'a': 1, 'b': 2}
>>> {**D, **{'b': 3, 'c': 4}, **dict(a=5, d=6)}
{'a': 5, 'b': 3, 'c': 4, 'd': 6}
```

Which—true sports fans will note—raises the number of dedicated dictionary merge operations from one to *three* since this book's prior edition. This bloat is especially grievous, given that it takes *just one line of simple code* to accomplish what the `update` method, the newer `**` unpacking, and the newest `|` union operator all redundantly do in their main use cases:

```
>>> D1 = dict(a=1, b=1)
>>> D2 = dict(a=2, c=2)

>>> for k in D2: D1[k] = D2[k]         # What update(),
>>> D1                                    # Did this really
{'a': 2, 'b': 1, 'c': 2}
```

Dictionary `|` "union" may be handy in limited contexts. But whether this justifies convoluting dictionaries with just one binary operator, just one of many set operations, and an operation that's almost entirely redundant with two earlier tools that were already almost entirely redundant with very simple code, is a riddle left to the reader to solve.

Intermission: Books Database

Let's take a break from the fine points and code a more concrete example of dictionaries at work. In honor of this book's quarter-century milestone, the following builds a simple in-memory editions database that maps edition *years* (the keys) to *titles* (the values). As coded, you fetch edition names by indexing on year strings:

```
>>> table = {'2024': 'Learning Python, 6th Edition',
             '2013': 'Learning Python, 5th Edition',
             '1999': 'Learning Python'}

>>> table['2024']
'Learning Python, 6th Edition'

>>> for year in table:
        print(year + '\t' + table[year])

2024    Learning Python, 6th Edition
2013    Learning Python, 5th Edition
1999    Learning Python
```

The last command uses a `for` loop, which we've used several times since its Chapter 4 preview. The full tale of the `for` isn't told until Chapter 13, but this particular loop simply iterates through each key in the table to print a tab-separated list of keys and their values (recall from Chapter 7 that `\t` in a Python string means horizontal tab).

Dictionaries aren't sequences like lists and strings, but if you need to step through the items in a dictionary, it's easy—either call the dictionary `keys` method or use the dictionary itself. Given a dictionary `D`, saying `for key` `in D` works the same as saying `for key in D.keys()`, and both use the *iteration protocol* that we'll expand on later in this book. Either way, you can index from *key* to *value* inside the `for` loop as you go, as this code does.

Mapping values to keys

Notice how the prior table maps years to titles, but not vice versa. If you want to map the other way—titles to years—you can either code the dictionary differently:

```
>>> table = {'Learning Python, 6th Edition': 2024,
             'Learning Python, 5th Edition': 2013,
             'Learning Python':              1999}

>>> table['Learning Python']
1999
```

Or use other dictionary methods like `items` that give searchable sequences (the `list` call is required in the following because `items` returns an iterable *view*, a topic coming up shortly):

```
>>> list(table.items())[:2]
[('Learning Python, 6th Edition', 2024), ('Learning Pyt

>>> [title for (title, year) in table.items() if year =
['Learning Python']
```

The last command here uses the list *comprehension* we explored earlier, as well as *tuple assignment* mentioned earlier and covered in this book's next part (synopsis: it unpacks items into variables). The combo scans the `(key, value)` pairs returned by the dictionary's `items` method, selecting keys for values matching a search target (`1999`).

The net effect of all this is to index *backward*—from value to key, instead of key to value. This is useful if you want to store data just once and map backward from values only rarely (searching through sequences like this is generally much slower than a direct key-to-value index, though not all programs need to care).

In fact, although dictionaries by nature map keys to values unidirectionally, there are multiple ways to map values back to keys with a bit of extra generalizable code:

```
>>> K = 'Learning Python'
>>> V = 1999
>>> table[K]              # Key = >Value (normal usage)
1999

>>> [key for (key, value) in table.items() if value ==
['Learning Python']

>>> [key for key in table.keys() if table[key] == V]
['Learning Python']
```

Note that both of the last two commands return a *list* of titles: in dictionaries, there's just *one* value per key, but there may be *many* keys per value if a given value is be stored under multiple keys, and a value might be a collection itself to represent many values per key. It's also possible to *invert* a dictionary for indexing values by zipping its values and keys—but *only* if its values are all immutable and don't appear in multiple keys:

```
>>> dict(zip(table.values(), table.keys()))[1999]
'Learning Python'
```

Sharp-eyed readers may notice that this yields a dictionary with *integer* keys, which works naturally per a tip in the next section. For more on this front, watch for a less lossy dictionary inversion function in the *mapattrs.py* example from "Example: Mapping Attributes to Inheritance Sources"—code that would surely stretch this preview past its breaking point if included here. For this chapter's purposes, let's wrap up by adding in a few pragmatic pieces to the dictionary puzzle.

Dictionary Usage Tips

Dictionaries are fairly straightforward tools once you get the hang of them, but here are a few additional pointers and reminders you should be aware of when using them:

Sequence operations don't work

Dictionaries are mappings, not sequences. Because they deal with keys and values and are ordered by insertion-time only, things like concatenation (an ordered joining of values) and slicing (extracting a contiguous section of values) simply don't apply—and Python raises an error if your code tries to use them on dictionaries.

Assigning to new indexes adds entries

Keys can be created when you write a dictionary literal (embedded in the code of the literal itself), or when you assign values to new keys of an existing dictionary object individually. The end result is the same.

Keys need not always be strings

Our examples so far have used strings as keys, but any other *immutable* objects work just as well. For instance, you can use integers as keys, which makes the dictionary look much like a list (when indexing, at least). Tuples may be used as dictionary keys too, allowing compound key values—such as dates and IP addresses—to have associated values. User-defined class instance objects (discussed in Part VI) can also be used as keys, as long as they define the proper methods; roughly, they need to tell Python that their values are "hashable" and thus won't change, as otherwise they would be useless as fixed keys. Mutable objects such as lists, sets, and other dictionaries don't work as keys because they may change, but are allowed as values.

The following sections delve deeper into these and other mysteries of dictionary-processing code.

Using dictionaries to simulate flexible lists: Integer keys

The last point in the preceding list is important enough to demonstrate with a few examples. When you use lists, it is illegal to assign to an offset that is off the end of the list:

```
>>> L = []
>>> L[99] = 'hack'
IndexError: list assignment index out of range
```

Although you can use repetition to preallocate as big a list as you'll need (e.g., `[0]*100`), you can also do something that looks similar with dictionaries that does not require such space allocations—and potential waste. By using integer keys, dictionaries can emulate lists that seem to grow on offset assignment:

```
>>> D = {}
>>> D[99] = 'hack'
>>> D[99]
'hack'
>>> D
{99: 'hack'}
```

Here, it looks as if `D` is a 100-item list, but it's really a dictionary with a single entry; the value of the key `99` is the string `'hack'`. You can access this structure with offsets much like a list, catching nonexistent keys with `get` or `in` tests if required, but you don't have to allocate space for all the positions to which you might need to assign values in the future. When used like this, dictionaries are like more flexible equivalents of lists:

```
>>> D[62] = 'code'
>>> D[30] = 'write'
>>> D[62]
'code'
>>> D
{99: 'hack', 62: 'code', 30: 'write'}
```

As another example, we might also employ integer keys in the first book *book-database* code we wrote earlier to avoid quoting the year, albeit at the expense of expressiveness (integer keys cannot contain nondigit characters):

```
>>> table = {2024: 'Learning Python, 6th Edition',
              …etc…
>>> table[2024]
'Learning Python, 6th Edition'
```

Using dictionaries for sparse data structures: Tuple keys

In a similar way, dictionary keys are also commonly leveraged to implement *sparse*—mostly empty—data structures, such as multidimensional arrays where only a few positions have values stored in them:

```
>>> Matrix = {}
>>> Matrix[(2, 3, 4)] = 88
>>> Matrix[(7, 8, 9)] = 99
>>>
>>> X = 2; Y = 3; Z = 4              # A ; separates s
>>> Matrix[(X, Y, Z)]
88
>>> Matrix
{(2, 3, 4): 88, (7, 8, 9): 99}
```

Here, we've used a dictionary to represent a three-dimensional array that is empty except for the two positions `(2,3,4)` and `(7,8,9)`. The keys are *tuples* that record the coordinates of nonempty slots. Rather than allocating a large and mostly empty three-dimensional matrix to hold these values, we can use a simple two-item dictionary. In this scheme, accessing an empty slot triggers a nonexistent key exception, as these slots are not physically stored:

```
>>> Matrix[(2,3,6)]
KeyError: (2, 3, 6)
```

Avoiding missing-key errors

Errors for nonexistent key fetches like the foregoing are common in sparse matrixes, but you probably won't want them to shut down your program. There are at least three ways to fill in a default value instead of getting such an error message—you can use the dictionary `get` method shown earlier to provide a default for keys that do not exist, test for keys ahead of time in `if` statements, or use a `try` statement to catch and recover from the error. Though straightforward, the last of these are previews of statement syntax we'll begin studying in Chapter 10:

```
>>> Matrix.get((2, 3, 4), 0)           # Exists: fetch
88
>>> Matrix.get((2, 3, 6), 0)           # Doesn't exist:
0

>>> if (2, 3, 6) in Matrix:            # Check for key
...     print(Matrix[(2, 3, 6)])       # See Chapters 1
... else:
...     print(0)
```

```
...
0

>>> try:
...     print(Matrix[(2, 3, 6)])        # Try to index
... except KeyError:                    # Catch and recc
...     print(0)                        # See Chapters 1
...
0
```

Of these, the `get` method is the most concise in terms of coding requirements, but the `if` and `try` statements are much more general in scope—as you'll start seeing for yourself soon in Chapter 10.

Nesting in dictionaries

As you can tell, dictionaries can play many roles in Python. In general, they can replace search data structures (because indexing by key is a search operation) and can represent many types of structured information. For example, dictionaries are one of many ways to describe the properties of an item in your program's domain; that is, they can serve the same role as "records" or "structs" in other language, and *JSON* content in language-neutral roles. The following, for example, fills out a dictionary describing a book, by assigning to new keys:

```
>>> rec = {}
>>> rec['title'] = 'Learning Python, 5th Edition'
>>> rec['year']  = 2013
>>> rec['isbn']  = '9781449355739'
>>>
>>> rec['year'], rec['isbn']
(2013, '9781449355739')
```

Especially when nested, though, Python's built-in data types allow us to easily represent *structured* information. The following again uses a dictionary to capture object properties, but it codes it all at once rather than assigning to each key separately, and nests a dictionary and list to represent structured property values:

```
>>> rec = {'title':  'Learning Python, 5th Edition',
           'date':   {'year': 2013, 'month': 'July'},
                 'isbns':  ['1449355730', '9781449355739']}
```

To fetch components of nested objects, simply string together indexing operations:

```
>>> rec['title']
'Learning Python, 5th Edition'
>>> rec['isbns']
['1449355730', '9781449355739']
>>> rec['isbns'][1]
'9781449355739'
>>> rec['date']['year']
2013
```

Although you'll learn in Part VI that *classes* (which group both data and logic) can sometimes be better in this record role, dictionaries are an easy-to-use tool for simpler requirements. For more on record representation choices, see also the upcoming sidebar "Why You Will Care: List Versus Dictionary Versus Set".

Also notice that while we've focused on a single "record" with nested data here, there's no reason we couldn't nest the record itself in a larger, enclosing *database* collection coded as a list or dictionary, though an external file or formal database interface often plays the role of top-level container in realistic programs. The following abstract snippets would both print a record's two-item `isbns` list if run live and provided with an `other` record omitted here:

```
db = []
db.append(rec)                # A list "database"
db.append(other)
db[0]['isbns']

db = {}
db['lp5e'] = rec               # A dictionary "database"
db['lp6e'] = other
db['lp5e']['isbns']
```

Later in the book you'll meet tools such as Python's `shelve`, which works much the same way, but automatically maps objects to and from files to make them permanent. Python objects really *can* be database records.

Dictionary key/value/item view objects

When we explored the dictionary `keys`, `values`, and `items` methods earlier, we wrapped their results in list calls for display at the REPL prompt. Technically, this is because these methods return *view objects* that produce results on demand, instead of physical lists. Displaying their raw values suggests as much, but collects their values anyhow:

```
>>> D = dict(program=1, script=2, app=3)
>>> D.keys()
dict_keys(['program', 'script', 'app'])
```

View objects are *iterables*, which we've seen simply means objects that generate result items one at a time, instead of producing the result list all at once in memory. Besides being iterable, dictionary views are *insertion* ordered, reflect future *changes* to the dictionary, and support *set* operations. On the other hand, because they are not lists, they do not directly support operations like indexing or the list `sort` method, and do not display as a normal list when printed.

We'll discuss the notion of iterables more formally in Chapter 14, but for our purposes here it's enough to know that we have to run the results of these three methods through the `list` built-in if we want to apply list operations or display their values as lists. For example:

```
>>> D = dict(a=1, b=2, c=3)
>>> D
{'a': 1, 'b': 2, 'c': 3}

>>> K = D.keys()                    # Makes a view objec
>>> K
dict_keys(['a', 'b', 'c'])
>>> list(K)                         # Force a real list
['a', 'b', 'c']

>>> V = D.values()                  # Ditto for values c
>>> V
dict_values([1, 2, 3])
>>> list(V)
[1, 2, 3]

>>> D.items()
```

```
dict_items([('a', 1), ('b', 2), ('c', 3)])
>>> list(D.items())
[('a', 1), ('b', 2), ('c', 3)]

>>> K[0]                              # List operations fc
TypeError: 'dict_keys' object is not subscriptable
>>> K.sort()
AttributeError: 'dict_keys' object has no attribute 'sc
>>> list(K)[0], list(K).sort()
('a', None)
```

Unlike lists, though, dictionary views don't take up *space* for their full results, and are not carved in stone when created—they *dynamically reflect future changes* made to the dictionary after the view object has been created:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'a': 1, 'b': 2, 'c': 3}

>>> K = D.keys()
>>> V = D.values()
>>> list(K), list(V)                 # Views maintain same
(['a', 'b', 'c'], [1, 2, 3])

>>> del D['b']                       # Change the dictionc
>>> D
{'a': 1, 'c': 3}

>>> list(K), list(V)                 # Reflected in any cu
(['a', 'c'], [1, 3])
```

That said, use cases for dynamic view morph are likely rare. In fact, apart from result displays at the interactive prompt, you probably won't even notice views very often in practice, because looping constructs in Python automatically force them to produce one result on each:

```
>>> for k in D.keys(): print(k)       # View iterators use
...                                   # But no need to cal
a
b
c
```

As we’ve seen, it’s also often unnecessary to call `keys` directly in such cases because dictionaries themselves provide *implicit* key iterators; for better or worse, `for k in D` is usually as much code as you need to type.

Dictionary views and sets

Though perhaps even more obscure than view objects in general, some dictionary views are also *set-like* and support set operations such as union and intersection that we used on true sets in Chapter 5. Specifically, views returned by the `keys` method are set-like, `values` views are not, and `items` views are if their `(key, value)` pairs are unique and hashable (immutable). This reflects logical symmetry: set items are unique and immutable just like dictionary keys, and sets themselves behave like unordered and valueless dictionaries (and are even coded in curly braces).

Here is what `keys` views look like when used in set operations (continuing the prior section’s session); as also shown, dictionary value views are never set-like, because their items are not necessarily unique or immutable:

```
>>> K, V
(dict_keys(['a', 'c']), dict_values([1, 3]))

>>> K | {'x': 4}                    # Keys (and some ite
{'x', 'a', 'c'}                     # Results are unorde

>>> V | {'x': 4}
TypeError: unsupported operand type(s) for |: 'dict_val
>>> V | {'x': 4}.values()
TypeError: unsupported operand type(s) for |: 'dict_val
```

Be careful not to confuse the `|` here with the *dictionary* union operation we met earlier in “Dictionary “Union” Operator”; when run on a view, `|` is not a full key-based dictionary merge. In set operations, views may be mixed with other views, sets, and dictionaries, and dictionaries are treated the same as their `keys` views:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D.keys() & D.keys()             # Intersect views
{'b', 'a', 'c'}
>>> D.keys() & {'b'}                # Intersect view anc
```

```
{'b'}
>>> D.keys() & {'b': 1}          # Intersect view anc
{'b'}
>>> D.keys() | {'b', 'c', 'd'}    # Union view and set
{'b', 'a', 'd', 'c'}
```

Items views are set-like too, but only if they are *hashable*—that is, if they contain only immutable objects:

```
>>> {'a': [1, 2]}.items() | {0, 1}     # Immutable valu
TypeError: unhashable type: 'list'

>>> D = {'a': 1}
>>> D.items()                          # Items set-like
dict_items([('a', 1)])
>>> D.items() | D.keys()               # Union view anc
{('a', 1), 'a'}
>>> D.items() | D                      # Dictionary tre
{('a', 1), 'a'}

>>> D.items() | {('c', 3), ('d', 4)}           # Set oj
{('a', 1), ('c', 3), ('d', 4)}

>>> dict(D.items() | {('c', 3), ('d', 4)})     # dict c
{'a': 1, 'c': 3, 'd': 4}
```

See Chapter 5's coverage of sets if you need a refresher on these operations. Their role in dictionary views is probably uncommon and may even be academic; but they work when helpful (and at least you now have a fighting chance when they crop up in sadistic final exams!). Here, let's wrap up with two more quick coding notes for dictionaries.

Sorting dictionary keys

First of all, keys lists must be sorted when the inherent insertion order doesn't meet your goals. Beware, though, that a common coding pattern for scanning a collection in sorted order won't work, because `keys` does not return a list:

```
>>> D = {'c': 3, 'b': 2, 'a': 1}
>>> D
{'c': 3, 'b': 2, 'a': 1}
```

```
>>> Ks = D.keys()                                   # Sorting
>>> Ks.sort()
AttributeError: 'dict_keys' object has no attribute 'sc
```

To work around this, you can either convert keys to a list manually, or use the `sorted` call, applied to lists earlier in this chapter, on either a `keys` view or the dictionary itself:

```
>>> Ks = list(D.keys())                             # Convert
>>> Ks.sort()                                       # And ther
>>> for k in Ks: print(k, D[k])
...
a 1
b 2
c 3

>>> Ks = D.keys()                                   # Or use s
>>> for k in sorted(Ks): print(k, D[k])             # sorted()
...                                                 # sorted()
a 1
b 2
c 3
```

Of these, using the dictionary's keys iterator is simplest and common, though also arguably implicit:

```
>>> for k in sorted(D): print(k, D[k])              # Or use s
...                                                 # dict ite
a 1
b 2
c 3
```

Dictionary magnitude comparisons

Finally, dictionaries can be compared for equality directly with `==`, which automatically compares each key/value pair and ignores insertion order:

```
>>> D1 = dict(a=1, b=2, c=3)
>>> D2 = dict(c=3, b=2, a=1)
>>> D1, D2
```

```
({'a': 1, 'b': 2, 'c': 3}, {'c': 3, 'b': 2, 'a': 1})
>>> D1 == D2, D1 != D2
(True, False)
```

Magnitude comparisons like `<` and `>` do not work on dictionaries themselves, though you can implement them by comparing key/value `items` views manually, if you also *sort* them to discount any insert-order difference (subtlety: a `>` on the `items` results directly runs the set *superset* test because the views are set-like—not greater-than!):

```
>>> D1 = dict(a=1, b=2, c=4)
>>> D2 = dict(c=3, b=2, a=1)                        # D1 > D
>>> D1 > D2
TypeError: '>' not supported between instances of 'dict

>>> list(D1.items()), list(D2.items())
([('a', 1), ('b', 2), ('c', 4)], [('c', 3), ('b', 2), (

>>> list(D1.items()) > list(D2.items())         # Fails:
False
>>> D1.items() > D2.items()                     # Fails:
False
>>> sorted(D1.items()) > sorted(D2.items())     # OK: sc
True
```

Because we'll revisit this near the end of the next chapter to reinforce them in the context of comparisons at large, we'll postpone further coverage here.

Chapter Summary

In this chapter, we explored the list and dictionary types—probably the two most common, flexible, and powerful collection types you will see and use in Python code. We learned that the list type supports positionally ordered collections of arbitrary objects, and that it may be freely nested and grown and shrunk on demand. The dictionary type is similar, but it stores items by key instead of by position and orders keys by insertion time only. Both lists and dictionaries are mutable, and so support a variety of in-place change operations not available for strings: for example, lists can be grown by slice assignment and `append` calls, and dictionaries by key assignment and `update`.

In the next chapter, we will wrap up our in-depth core object-type tour by looking at tuples and files. After that, we'll move on to statements that code the logic that processes our objects, taking us another step toward writing complete programs. Before we tackle those topics, though, here are some chapter quiz questions to review what you've learned.

Test Your Knowledge: Quiz

1. Name two ways to build a list containing five integer zeros.
2. Name two ways to build a dictionary with two keys, `'a'` and `'b'`, each having an associated value of `0`.
3. Name four operations that change a list object in place.
4. Name four operations that change a dictionary object in place.
5. Why might you use a dictionary instead of a list?

Test Your Knowledge: Answers

1. A literal expression like `[0, 0, 0, 0, 0]` and a repetition expression like `[0] * 5` will each create a list of five zeros. In practice, you might also build one up with a loop that starts with an empty list and appends `0` to it in each iteration, with `L.append(0)`. A list comprehension like `[0 for i in range(5)]` could work here, too, but this is more work than you need to do for this answer.
2. A literal expression such as `{'a': 0, 'b': 0}` or a series of assignments like `D = {}`, `D['a'] = 0`, and `D['b'] = 0` would create the desired dictionary. You can also use the newer and simpler-to-code `dict(a=0, b=0)` keyword form, or the more flexible `dict([('a', 0), ('b', 0)])` key/value sequences form. Because all the values are the same, you can also use the special form `dict.fromkeys('ab', 0)`, and dictionary comprehension like `{k: 0 for k in 'ab'}` suffices too, though again, this may be overkill here.
3. The `append` and `extend` methods grow a list in place, the `sort` and `reverse` methods order and reverse lists, the `insert` method inserts an item at an offset, the `remove` and `pop` methods delete from a list by value and by position, the `del` statement deletes an item or slice, and

index and slice assignment statements replace an item or entire section. Pick any four of these for the quiz.

4. Dictionaries are primarily changed by *assignment* to a new or existing key, which creates or changes the key's associated value in the table. Also, the `del` statement deletes a key's entry, the dictionary `update` method merges one dictionary into another in place, and `D.pop(key)` removes a key and returns the value it had. Dictionaries also have other, more exotic in-place change methods presented tersely or not at all in this chapter, such as `popitem` and `setdefault`; see reference resources for more details.
5. This question is a bit unfair, given that the following sidebar gives the answer, but you may have already figured this out on your own. Dictionaries are generally better when the data is labeled (a record with field names, for example); lists are best suited to collections of unlabeled items (such as all the files in a directory). Dictionary lookup is also usually quicker than searching a list, though this might vary per program and Python.

WHY YOU WILL CARE: LIST VERSUS DICTIONARY VERSUS SET

With all the objects in Python's core types arsenal, some readers may be puzzled over the choice between lists and dictionaries. In short, although both are flexible collections of other objects, lists assign items to *positions*, and dictionaries assign them to more mnemonic *keys*. Because of this, dictionary data often carries more meaning to human readers. For example, a nested list structure can always be used to record info:

```
>>> L = ['Pat', 40.5, ['dev', 'mgr']]  # List-based "re
>>> L[0]
'Pat'
>>> L[1]                               # Positions/numb
40.5
>>> L[2][1]
'mgr'
```

For some types of data, the list's access-by-position makes sense—a list of employees in a company, the files in a directory, or numeric matrixes, for example. But a more symbolic record like this may be more meaningfully coded as a dictionary, with labeled fields replacing field positions:

```
>>> D = {'name': 'Pat', 'age': 40.5, 'jobs': ['dev', 'm
>>> D['name']
'Pat'
>>> D['age']                           # Dictionary-bas
40.5
>>> D['jobs'][1]                       # Names mean mor
'mgr'
```

For variety, here is the same record recoded with `dict` and keywords, which may seem even more readable to some human readers:

```
>>> D = dict(name='Pat', age=40.5, jobs=['dev', 'mgr'])
>>> D['name']
'Pat'
>>> D['jobs'].remove('mgr')
>>> D
{'name': 'Pat', 'age': 40.5, 'jobs': ['dev']}
```

In practice, dictionaries tend to be best for data with labeled components, as well as structures that can benefit from quick, direct lookups by name, instead of slower linear (left-to-right) searches. As we've seen, they also may be better for sparse collections and collections that grow flexibly.

Python programmers also have access to the *sets* we studied in Chapter 5, which are much like the keys of a valueless dictionary; they don't map keys to values, but can often be used like dictionaries for fast lookups when there is no associated value, especially in search routines:

```
>>> D = {}
>>> D['state1'] = True                   # A visited-stat
>>> 'state1' in D
True
>>> S = set()
>>> S.add('state1')                      # Same, but with
>>> 'state1' in S
True
```

Watch for a rehash of this record representation thread in the next chapter, where you'll see how *tuples* and *named tuples* compare to dictionaries in this role, as well as in Chapter 27, where you'll learn how user-defined *classes* factor into this picture, combining both data and logic to process it.

---

1 In practice, you won't see many lists written out like this in list-processing programs. It's more common to see code that processes lists constructed dynamically (at runtime), from user inputs, file contents, and so on. In fact, although it's important to master literal syntax, many data structures in Python are built by running program code at runtime.

2 This description requires elaboration when the value and the slice being assigned *overlap*: `L[2:5]=L[3:6]`, for instance, works fine because the value to be inserted is fetched *before* the deletion happens on the left. Hence, slice assignment is really a *fetch* + *delete* + *insert*, but the fetch part matters too rarely to make the marquee.

3 Unlike `+` concatenation, `append` doesn't have to generate new objects, so it's usually faster than `+` too. You can also mimic `append` with the clever slice assignments of the prior section: `L[len(L):]=[X]` is like `L.append(X)`, and `L[:0]=[X]` is like appending at the front of a list. Both delete an empty slice and insert `X`, changing `L` in place quickly, like `append`. Both are arguably more

complex than list methods, though. For instance, `L.insert(0, X)` can also append an item to the front of a list, and seems noticeably more mnemonic. `L.insert(len(L), X)` inserts one object at the end too, but unless you like typing, you might as well use `L.append(X)` !

[4] As for lists, you might not see dictionaries coded in full using literals very often—programs rarely know all their data before they are run, and more typically extract it dynamically from users, files, and so on. Lists and dictionaries are grown in different ways, though. In the next section you'll see that you often build up dictionaries by assigning to new keys at runtime; this approach fails for lists, which are commonly grown with `append` or `extend` instead.