CHAPTER 8

Lists and Dictionaries

This chapter presents the list and dictionary object types, both of which are collections of other objects. These two types are the main workhorses in almost all Python scripts. As you'll see, both types 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 types, you can build up and process arbitrarily rich information structures in your scripts.

Lists

The next stop on our built-in object 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 most 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 (i.e., they are sequences).

Accessed by offset
: Just as with strings, you can fetch a component object out of 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 can contain any sort of object, not just one-character strings (they're heterogeneous). Because lists can contain other complex 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 can be changed in-place (they're mutable), 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 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). Fetching an item from a Python list is about as fast as indexing a C array; in fact, lists really are C arrays inside the standard Python 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 (unless you request a copy explicitly).

Table 8-1 summarizes common and representative list object operations. As usual, for the full story, see the Python standard library manual, or run a `help(list)` or `dir(list)` call interactively for a full list of list methods—you can pass in a real list, or the word `list`, which is the name of the list data type.

Table 8-1. Common list literals and operations

Operation	Interpretation
`L1 = []`	An empty list
`L2 = [0, 1, 2, 3]`	Four items: indexes 0..3
`L3 = ['abc', ['def', 'ghi']]`	Nested sublists

`L2[i]`
`L3[i][j]`
`L2[i:j]`
`len(L2)`
	Index, index of index, slice, length

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

Table 8-1. *Common list literals and operations (continued)*

Operation	Interpretation

`for x in L2`
`3 in L2`
	Iteration, membership

`L2.append(4)`
`L2.extend([5,6,7])`
`L2.sort( )`
`L2.index(1)`
`L2.insert(I, X)`
`L2.reverse( )`
	Methods: grow, sort, search, insert, reverse, etc.

`del L2[k]`
`del L2[i:j]`
`L2.pop( )`
`L2.remove(2)`
`L2[i:j] = []`
	Shrinking

`L2[i] = 1`
`L2[i:j] = [4,5,6]`
	Index assignment, slice assignment

`range(4)`
`xrange(0, 4)`
	Make lists/tuples of integers

`L4 = [x**2 for x in range(5)]`	List comprehensions (Chapters 13 and 17)

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 `L2` 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).[*]

Many of the operations in Table 8-1 should look familiar, as they are the same sequence operations we put to work on strings—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). Lists get these tools for change operations because they are a mutable object type.

Lists in Action

Perhaps 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

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. In fact, lists

[*] 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). In fact, although it's important to master literal syntax, most data structures in Python are built by running program code at runtime.

respond to all of the general sequence operations we used on strings in the prior chapter:

```
% python
>>> len([1, 2, 3])                      # Length
3
>>> [1, 2, 3] + [4, 5, 6]               # Concatenation
[1, 2, 3, 4, 5, 6]
>>> ['Ni!'] * 4                         # Repetition
['Ni!', 'Ni!', 'Ni!', 'Ni!']
>>> 3 in [1, 2, 3]                      # Membership
True
>>> for x in [1, 2, 3]: print x,        # Iteration
...
1 2 3
```

We will talk more about `for` iteration and the `range` built-ins in Chapter 13 because they are related to statement syntax. In short, `for` loops step through items in a sequence from left to right, executing one or more statements for each item. The last entry in Table 8-1, which is list comprehensions, are covered in Chapter 13, and expanded on in Chapter 17; as introduced in Chapter 4, they are a way to build a list by applying an expression to each item in a sequence, in a single step.

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

Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings. However, the result of indexing a list is whatever type of object lives at the offset you specify, while slicing a list always returns a new list:

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[2]                                # Offsets start at zero
'SPAM!'
>>> L[-2]                               # Negative: count from the right
'Spam'
>>> L[1:]                               # Slicing fetches sections
['Spam', 'SPAM!']
```

One note 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:

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

Notice in the preceding interaction that lists can naturally span multiple lines if you want them to because they are contained by a pair of brackets (more on syntax in the next part of the book). Later in this chapter, you'll also see a dictionary-based matrix representation. For high-powered numeric work, the NumPy extension mentioned in Chapter 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, the operations in this section all modify the list object directly, without forcing you to 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

When using a list, you can change its contents by assigning to a particular item (offset), or an entire section (slice):

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[1] = 'eggs'                  # Index assignment
>>> L
['spam', 'eggs', 'SPAM!']
>>> L[0:2] = ['eat', 'more']       # Slice assignment: delete+insert
>>> L                              # Replaces items 0,1
['eat', 'more', 'SPAM!']
```

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 C and most other languages: Python replaces the object reference at the designated offset with a new one.

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 object to the right of the `=` are inserted into the list on the left, at the place where the old slice was deleted.[*]

This isn't what really happens, but it tends to help clarify why the number of items inserted doesn't have to match the number of items deleted. For instance, given a list `L` that has the value `[1,2,3]`, the assignment `L[1:2]=[4,5]` sets `L` to the list `[1,4,5,3]`. Python first deletes the `2` (a one-item slice), then inserts the `4` and `5` where the deleted `2` used to be. This also explains why `L[1:2]=[]` is really a deletion operation—Python deletes the slice (the item at offset 1), and then inserts nothing.

In effect, slice assignment replaces an entire section, or "column," all at once. 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 usually more straightforward ways to replace, insert, and delete (concatenation, and the `insert`, `pop`, and `remove` list methods, for example), which Python programmers tend to prefer in practice.

List method calls

Like strings, Python list objects also support type-specific method calls:

```
>>> L.append('please')                # Append method call
>>> L
['eat', 'more', 'SPAM!', 'please']
>>> L.sort( )                          # Sort list items ('S' < 'e')
>>> L
['SPAM!', 'eat', 'more', 'please']
```

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

[*] This description needs 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.

Perhaps the most commonly used list method is append, which simply tacks a single item (object reference) onto the end of the list. Unlike concatenation, append expects you to pass in a single object, not a list. The effect of L.append(X) is similar to L+[X], but while the former changes L in-place, the latter makes a new list.[*] Another commonly seen method, sort, orders a list in-place; by default, it uses Python standard comparison tests (here, string comparisons), and sorts in ascending order. You can also pass in a comparison function of your own to sort.

In Python 2.5 and earlier, comparisons of differently typed objects (e.g., a string and a list) work—the language defines a fixed ordering among different types, which is deterministic, if not aesthetically pleasing. That is, this ordering is based on the names of the types involved: all integers are less than all strings, for example, because "int" is less than "str". Comparisons never automatically convert types, except when comparing numeric type objects.

In Python 3.0, this may change: comparison of mixed types is scheduled to raise an exception instead of falling back on the fixed cross-type ordering. Because sorting uses comparisons internally, this means that [1, 2, 'spam'].sort( ) succeeds in Python 2.x, but will raise an exception as of Python 3.0. See the 3.0 release notes for more details.

One warning here: beware that append and sort change the associated list object in-place, but don't return the list as a result (technically, they both return a value called None). If you say something like 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 attributes such as append and sort, objects are changed as a side effect, so there's no reason to reassign.

Like strings, lists have other methods that perform other specialized operations. For instance, reverse reverses the list in-place, and the extend and pop methods insert multiple items at the end, and delete an item from the end of the list, respectively:

```
>>> L = [1, 2]
>>> L.extend([3,4,5])              # Append multiple items
>>> L
[1, 2, 3, 4, 5]
>>> L.pop( )                       # Delete and return last item
5
>>> L
[1, 2, 3, 4]
>>> L.reverse( )                   # In-place reversal
>>> L
[4, 3, 2, 1]
```

[*] Unlike + concatenation, append doesn't have to generate new objects, so it's usually faster. You can also mimic append with clever slice assignments: 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.

In some types of programs, the list `pop` method used here is often 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]
```

Although not shown here, the `pop` method also accepts an optional offset of the item to be deleted and returned (the default is the last item). Other list methods remove an item by value (`remove`), insert an item at an offset (`insert`), search for an item's offset (`index`), and more; see other documentation sources, or experiment with these calls interactively on your own to learn more.

Other common list operations

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

```
>>> L
['SPAM!', 'eat', 'more', 'please']
>>> del L[0]                       # Delete one item
>>> L
['eat', 'more', 'please']
>>> del L[1:]                      # Delete an entire section
>>> L                              # Same as L[1:] = []
['eat']
```

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 in the specified slot, rather than deleting it:

```
>>> L = ['Already', 'got', 'one']
>>> L[1:] = []
>>> L
['Already']
>>> L[0] = []
>>> L
[[]]
```

Although all the operations just discussed are typical, there are additional list methods and operations not illustrated here (including methods for inserting and searching). For a comprehensive and up-to-date list of type tools, you should always consult

Python's manuals, Python's `dir` and `help` functions (which we first met in Chapter 4), or the *Python Pocket Reference* (O'Reilly), and other reference texts described in the Preface.

I'd also like to remind you one more time that all the in-place change operations discussed here work only for mutable objects: they won't work on strings (or tuples, discussed in the next chapter), no matter how hard you try. Mutability is an inherent property of each object type.

Dictionaries

Apart from lists, *dictionaries* are perhaps the most flexible built-in data type in Python. If you think of lists as ordered collections of objects, you can think of dictionaries as unordered collections; the chief distinction is that in dictionaries, items are stored and fetched by *key*, instead of by positional offset.

Being a built-in type, dictionaries can replace many of the searching algorithms and data structures you might have to implement manually in lower-level languages—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 represent sparse (mostly empty) data structures, and much more. Here's a rundown of their main properties. Python dictionaries are:

Accessed by key, not offset
Dictionaries are sometimes called *associative arrays* or *hashes*. They associate a set of values with 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 in a list, but the index takes the form of a key, not a relative offset.

Unordered collections of arbitrary objects
Unlike in a list, items stored in a dictionary aren't kept in any particular order; in fact, Python randomizes their left-to-right order to provide quick lookup. 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 contain lists, other dictionaries, and so on).

Of the category mutable mapping
Dictionaries can be changed 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 unordered collections, operations that depend on a fixed positional order (e.g., concatenation, slicing) don't make sense. Instead, dictionaries are the only built-in representatives of the mapping type category (objects that map keys to values).

Tables of object references (hash tables)

If lists are arrays of object references that support access by position, dictionaries are unordered 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, dictionaries store object references (not copies).

Table 8-2 summarizes some of the most common and representative dictionary operations (again, see the library manual or run a `dir(dict)` or `help(dict)` call for a complete list—`dict` is the name of the type). When coded as a literal expression, a dictionary is written as a series of `key:value` pairs, separated by commas, enclosed in curly braces.* An empty dictionary is an empty set of braces, and dictionaries can be nested by writing one as a value inside another dictionary, or within a list or tuple.

Table 8-2. Common dictionary literals and operations

Operation	Interpretation
`D1 = {}`	Empty dictionary
`D2 = {'spam': 2, 'eggs': 3}`	Two-item dictionary
`D3 = {'food': {'ham': 1, 'egg': 2}}`	Nesting

`D2['eggs']`
`D3['food']['ham']`
	Indexing by key

`D2.has_key('eggs')`
`'eggs' in D2`
`D2.keys( )`
`D2.values( )`
`D2.copy( )`
`D2.get(key, default)`
`D2.update(D1)`
`D2.pop(key)`
	Methods: membership test, keys list, values list, copies, defaults, merge, delete, etc.

`len(D1)`	Length (number of stored entries)

`D2[key] = 42`
`del D2[key]`
	Adding/changing keys, deleting keys

`D4 = dict.fromvalues(['a', 'b'])`
`D5 = dict(zip(keyslist, valslist))`
`D6 = dict(name='Bob', age=42)`
	Alternative construction techniques

Dictionaries in Action

As Table 8-2 suggests, dictionaries are indexed by key, and nested dictionary entries are referenced by a series of indexes (keys in square brackets). When Python creates

* As with lists, you won't often see dictionaries constructed using literals. Lists and dictionaries are grown in different ways, though. As you'll see in the next section, dictionaries are typically built up by assigning to new keys at runtime; this approach fails for lists (lists are grown with append instead).

a dictionary, it stores its items in any left-to-right order it chooses; to fetch a value back, you supply the key with which it is associated. Let's go back to the interpreter to get a feel for some of the dictionary operations in Table 8-2.

Basic Dictionary Operations

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

```
% python
>>> d2 = {'spam': 2, 'ham': 1, 'eggs': 3}     # Make a dictionary
>>> d2['spam']                                # Fetch a value by key
2
>>> d2                                        # Order is scrambled
{'eggs': 3, 'ham': 1, 'spam': 2}
```

Here, the dictionary is assigned to the variable `d2`; the value of the key `'spam'` 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: the left-to-right order of keys in a dictionary will almost always be different from what you originally typed. This is on purpose—to implement fast key lookup (a.k.a. hashing), keys need to be randomized in memory. That's why operations that assume a fixed left-to-right order (e.g., slicing, concatenation) do not apply to dictionaries; you can fetch values only by key, not by position.

The built-in `len` function works on dictionaries, too; it returns the number of items stored in the dictionary or, equivalently, the length of its keys list. The dictionary `has_key` method and the `in` membership operator allow you to test for key existence, and the `keys` method returns all the keys in the dictionary, collected in a list. The latter of these can be useful for processing dictionaries sequentially, but you shouldn't depend on the order of the keys list. Because the `keys` result is a normal list, however, it can always be sorted if order matters:

```
>>> len(d2)                         # Number of entries in dictionary
3
>>> d2.has_key('ham')               # Key membership test
True
>>> 'ham' in d2                     # Key membership test alternative
True
>>> d2.keys( )                      # Create a new list of my keys
['eggs', 'ham', 'spam']
```

Notice the third expression in this listing. As mentioned earlier, the `in` membership test used for strings and lists also works on dictionaries—it checks whether a key is stored in the dictionary, like the `has_key` method call of the prior line. Technically, this works because dictionaries define *iterators* that step through their keys lists. Other types provide iterators that reflect their common uses; files, for example, have iterators that read line by line. We'll discuss iterators further in Chapters 17 and 24.

Later in this chapter and book, you'll learn about two alternative ways to build dictionaries, demonstrated at the end of Table 8-2: you can pass zipped lists of key/value tuples or keyword function arguments to the new `dict` call (really, a type constructor). We'll explore keyword arguments in Chapter 16. We'll also discuss the `zip` function in Chapter 13; it's a way to construct a dictionary from key and value lists in a single call. If you cannot predict the set of keys and values in your code, for instance, you can always build them up as lists and zip them together dynamically.

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 `'ham'`). All collection data types in Python can nest inside each other arbitrarily:

```
>>> d2['ham'] = ['grill', 'bake', 'fry']             # Change entry
>>> d2
{'eggs': 3, 'spam': 2, 'ham': ['grill', 'bake', 'fry']}

>>> del d2['eggs']                                   # Delete entry
>>> d2
{'spam': 2, 'ham': ['grill', 'bake', 'fry']}

>>> d2['brunch'] = 'Bacon'                           # Add new entry
>>> d2
{'brunch': 'Bacon', 'spam': 2, 'ham': ['grill', 'bake', 'fry']}
```

As with lists, assigning to an existing index in a dictionary changes its associated value. Unlike with lists, however, whenever you assign a *new* dictionary key (one that hasn't been assigned before), you create a new entry in the dictionary, as was done in the previous example for the key `'brunch.'` This doesn't work for lists because Python considers an offset beyond the end of a list out of bounds and throws an error. 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 tools. For instance, the dictionary `values` and `items` methods return lists of the dictionary's values and `(key,value)` pair tuples, respectively:

```
>>> d2 = {'spam': 2, 'ham': 1, 'eggs': 3}
>>> d2.values()
 [3, 1, 2]
>>> d2.items()
 [('eggs', 3), ('ham', 1), ('spam', 2)]
```

Such lists are useful in loops that need to step through dictionary entries one by one. 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:

```
>>> d2.get('spam')                          # A key that is there
2
>>> d2.get('toast')                         # A key that is missing
None
>>> d2.get('toast', 88)
88
```

The update method provides something similar to concatenation for dictionaries. It merges the keys and values of one dictionary into another, blindly overwriting values of the same key:

```
>>> d2
{'eggs': 3, 'ham': 1, 'spam': 2}
>>> d3 = {'toast':4, 'muffin':5}
>>> d2.update(d3)
>>> d2
{'toast': 4, 'muffin': 5, 'eggs': 3, 'ham': 1, 'spam': 2}
```

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:

```
# pop a dictionary by key

>>> d2
{'toast': 4, 'muffin': 5, 'eggs': 3, 'ham': 1, 'spam': 2}
>>> d2.pop('muffin')
5
>>> d2.pop('toast')                         # Delete and return from a key
4
>>> d2
{'eggs': 3, 'ham': 1, 'spam': 2}

# pop a list by position

>>> L = ['aa', 'bb', 'cc', 'dd']
>>> L.pop()                                 # Delete and return from the end
'dd'
>>> L
['aa', 'bb', 'cc']
>>> L.pop(1)                                # Delete from a specific position
'bb'
>>> L
['aa', 'cc']
```

Dictionaries also provide a copy method; we'll discuss this in the next chapter, as it's a way to avoid the potential side effects of shared references to the same dictionary. In fact, dictionaries come with many more methods than those listed in Table 8-2; see the Python library manual, or other documentation sources for a comprehensive list.

A Languages Table

Let's look at a more realistic dictionary example. The following example creates a table that maps programming language names (the keys) to their creators (the values). You fetch creator names by indexing on language names:

```
>>> table = {'Python':  'Guido van Rossum',
...          'Perl':    'Larry Wall',
...          'Tcl':     'John Ousterhout' }
...
>>> language = 'Python'
>>> creator  = table[language]
>>> creator
'Guido van Rossum'

>>> for lang in table.keys( ):
...     print lang, '\t', table[lang]
...
Tcl     John Ousterhout
Python  Guido van Rossum
Perl    Larry Wall
```

The last command uses a for loop, which we haven't covered in detail yet. If you aren't familiar with for loops, this command simply iterates through each key in the table, and prints a tab-separated list of keys and their values. We'll learn more about for loops in Chapter 13.

Because dictionaries aren't sequences, you can't iterate over them directly with a for statement in the way you can with strings and lists. But, if you need to step through the items in a dictionary, it's easy: calling the dictionary keys method returns a list of all stored keys, which you can iterate through with a for. If needed, you can index from key to value inside the for loop, as was done in this code.

In fact, Python also lets you step through a dictionary's keys list without actually calling the keys method in most for loops. For any dictionary D, saying for key in D: works the same as saying the complete for key in D.keys( ):. This is really just another instance of the iterators mentioned earlier, which allow the in membership operator to work on dictionaries as well (more on iterators later in this book).

Dictionary Usage Notes

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 there's no notion of ordering among their items, things like concatenation (an ordered joining), and slicing (extracting a contiguous section) simply don't apply. In fact, Python raises an error when your code runs if you try to do such things.
- *Assigning to new indexes adds entries.* Keys can be created when you write a dictionary literal (in which case they are embedded in the literal itself), or when you assign values to new keys of an existing dictionary object. The end result is the same.
- *Keys need not always be strings.* Our examples used strings as keys, but any other *immutable* objects (i.e., not lists) 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 are sometimes used as dictionary keys too, allowing for compound key values. Class instance objects (discussed in Part VI) can be used as keys too, as long as they have the proper protocol methods; roughly, they need to tell Python that their values won't change, as otherwise they would be useless as fixed keys.

Using dictionaries to simulate flexible lists

The last point in the prior 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] = 'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
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, which does not require such space allocations. By using integer keys, dictionaries can emulate lists that seem to grow on offset assignment:

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

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 'spam'. You can access this structure with offsets much like a list, but you don't have to allocate space for all the positions you might ever need to assign values to in the future. When used like this, dictionaries are like more flexible equivalents of lists.

Using dictionaries for sparse data structures

In a similar way, dictionary keys are also commonly leveraged to implement *sparse* data structures—for example, 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                # ; separates statements
>>> 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, 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)]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: (2, 3, 6)
```

Avoiding missing-key errors

Errors for nonexistent key fetches 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 test for keys ahead of time in if statements, use a try statement to catch and recover from the exception explicitly, or simply use the dictionary get method shown earlier to provide a default for keys that do not exist:

```
>>> if Matrix.has_key((2,3,6)):      # Check for key before fetch
...     print Matrix[(2,3,6)]
... else:
...     print 0
...
0
```

```
>>> try:
...     print Matrix[(2,3,6)]        # Try to index
... except KeyError:                 # Catch and recover
...     print 0
...
0
>>> Matrix.get((2,3,4), 0)           # Exists; fetch and return
88
>>> Matrix.get((2,3,6), 0)           # Doesn't exist; use default arg
0
```

Of these, the get method is the most concise in terms of coding requirements; we'll study the if and try statements in more detail later in this book.

Using dictionaries as "records"

As you can see, 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 languages.

This example fills out a dictionary by assigning to new keys over time:

```
>>> rec = {}
>>> rec['name'] = 'mel'
>>> rec['age']  = 45
>>> rec['job']  = 'trainer/writer'
>>>
>>> print rec['name']
mel
```

Especially when nested, Python's built-in data types allow us to easily represent structured information. This example 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 list and a dictionary to represent structured property values:

```
>>> mel = {'name': 'Mark',
...        'jobs': ['trainer', 'writer'],
...        'web':  'www.rmi.net/~lutz',
...        'home': {'state': 'CO', 'zip':80513}}
...
```

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

```
>>> mel['name']
'Mark'
>>> mel['jobs']
['trainer', 'writer']
>>> mel['jobs'][1]
'writer'
>>> mel['home']['zip']
80513
```

Other ways to make dictionaries

Finally, note that because dictionaries are so useful, more ways to build them have emerged over time. In Python 2.3 and later, for example, the last two calls to the `dict` constructor in the following have the same effect as the literal and key-assignment forms above them:

```
{'name': 'mel', 'age': 45}              # Traditional literal expression

D = {}                                  # Assign by keys dynamically
D['name'] = 'mel'
D['age']  = 45

dict(name='mel', age=45)                # Keyword argument form

dict([('name', 'mel'), ('age', 45)])    # Key/value tuples form
```

All four of these forms create the same two-key dictionary:

- 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 keyword form is less code to type than literals, but it requires all keys to be strings.
- The last form here is useful if you need to build up keys and values as sequences at runtime.

Why You Will Care: Dictionary Interfaces

Besides being a convenient way to store information by key in your programs, some Python extensions also present interfaces that look like and work the same as dictionaries. For instance, Python's interface to DBM access-by-key files looks much like a dictionary that must be opened. Strings are stored and fetched using key indexes:

```
import anydbm
file = anydbm.open("filename") # Link to file
file['key'] = 'data'           # Store data by key
data = file['key']             # Fetch data by key
```

Later, you'll see that you can store entire Python objects this way, too, if you replace `anydbm` in the preceding code with `shelve` (shelves are access-by-key databases of persistent Python objects). For Internet work, Python's CGI script support also presents a dictionary-like interface. A call to `cgi.FieldStorage` yields a dictionary-like object with one entry per input field on the client's web page:

```
import cgi
form = cgi.FieldStorage( )      # Parse form data
if form.has_key('name'):
    showReply('Hello, ' + form['name'].value)
```

All of these (and dictionaries) are instances of mappings. Once you learn dictionary interfaces, you'll find that they apply to a variety of built-in tools in Python.

As suggested near the end of Table 8-2, the last form 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).

Provided all the key's values are the same, you can also initialize a dictionary with this special form—simply pass in a list of keys and an initial value for all of them (the default is `None`):

```
>>> 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, flexible, and dynamic Python programs.

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, grown and shrunk on demand, and more. The dictionary type is similar, but it stores items by key instead of by position, and does not maintain any reliable left-to-right order among its items. 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 `append` calls, and dictionaries by assignment to new keys.

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.

Chapter 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.

Quiz 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: `L.append(0)`. A list comprehension (`[0 for i in range(5)]`) could work here too, but this is more work than you need to do.
2. A literal expression such as `{'a': 0, 'b': 0}`, or a series of assignments like `D = []`, `D['a'] = 0`, `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. Or, because all the keys are the same, you can use the special form `dict.fromkeys(['a', 'b'], 0)`.
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 entry 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 not listed in this chapter, such as `setdefault`; see reference sources for more details.