Beginning\_Python\_Magnus\_04

CHAPTER 4

Dictionaries: When Indices

Won’t Do

You’ve seen that lists are useful when you want to group values into a structure and refer to each value by

number. In this chapter, you learn about a data structure in which you can refer to each value by name. This

type of structure is called a mapping. The only built-in mapping type in Python is the dictionary. The values

in a dictionary don’t have any particular order but are stored under a key, which may be a number, a string,

or even a tuple.

Dictionary Uses

The name dictionary should give you a clue about the purpose of this structure. An ordinary book is made

for reading from start to finish. If you like, you can quickly open it to any given page. This is a bit like a

Python list. On the other hand, dictionaries—both real ones and their Python equivalent—are constructed

so that you can look up a specific word (key) easily to find its definition (value).

A dictionary is more appropriate than a list in many situations. Here are some examples of uses of

Python dictionaries:

•Representing the state of a game board, with each key being a tuple of coordinates

•Storing file modification times, with file names as keys

•A digital telephone/address book

Let’s say you have a list of people.

>>> names = ['Alice', 'Beth', 'Cecil', 'Dee-Dee', 'Earl']

What if you wanted to create a little database where you could store the telephone numbers of these

people—how would you do that? One way would be to make another list. Let’s say you’re storing only their

four-digit extensions. Then you would get something like this:

>>> numbers = ['2341', '9102', '3158', '0142', '5551']

Once you’ve created these lists, you can look up Cecil’s telephone number as follows:

>>> numbers[names.index('Cecil')]

'3158'

It works, but it’s a bit impractical. What you really would want to do is something like the following:

>>> phonebook['Cecil']

'3158'

Guess what? If phonebook is a dictionary, you can do just that.

Creating and Using Dictionaries

Dictionaries are written like this:

phonebook = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}

Dictionaries consist of pairs (called items) of keys and their corresponding values. In this example, the names

are the keys, and the telephone numbers are the values. Each key is separated from its value by a colon (:),

the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary

(without any items) is written with just two curly braces, like this: {}.

■■Note Keys are unique within a dictionary (and any other kind of mapping). Values do not need to be unique

within a dictionary.

The dict Function

You can use the dict function1 to construct dictionaries from other mappings (for example, other

dictionaries) or from sequences of (key, value) pairs.

>>> items = [('name', 'Gumby'), ('age', 42)]

>>> d = dict(items)

>>> d

{'age': 42, 'name': 'Gumby'}

>>> d['name']

'Gumby'

It can also be used with keyword arguments, as follows:

>>> d = dict(name='Gumby', age=42)

>>> d

{'age': 42, 'name': 'Gumby'}

Although this is probably the most useful application of dict, you can also use it with a mapping argument

to create a dictionary with the same items as the mapping. (If used without any arguments, it returns a new

empty dictionary, just like other similar functions such as list, tuple, and str.) If the other mapping is

a dictionary (which is, after all, the only built-in mapping type), you can use the dictionary method copy

instead, as described later in this chapter.

Basic Dictionary Operations

The basic behavior of a dictionary in many ways mirrors that of a sequence.

•len(d) returns the number of items (key-value pairs) in d.

•d[k] returns the value associated with the key k.

•d[k] = v associates the value v with the key k.

•del d[k] deletes the item with key k.

•k in d checks whether there is an item in d that has the key k.

Although dictionaries and lists share several common characteristics, there are some important distinctions:

Key types: Dictionary keys don’t have to be integers (though they may be). They

may be any immutable type, such as floating-point (real) numbers, strings, or

tuples.

Automatic addition: You can assign a value to a key, even if that key isn’t in the

dictionary to begin with; in that case, a new item will be created. You cannot

assign a value to an index outside the list’s range (without using append or

something like that).

Membership: The expression k in d (where d is a dictionary) looks for a key,

not a value. The expression v in l, on the other hand (where l is a list) looks

for a value, not an index. This may seem a bit inconsistent, but it is actually quite

natural when you get used to it. After all, if the dictionary has the given key,

checking the corresponding value is easy.

■■Tip Checking for key membership in a dictionary is more efficient than checking for membership in a list.

The difference is greater the larger the data structures are.

The first point—that the keys may be of any immutable type—is the main strength of dictionaries. The

second point is important, too. Just look at the difference here:

>>> x = []

>>> x[42] = 'Foobar'

Traceback (most recent call last):

File "<stdin>", line 1, in ?

IndexError: list assignment index out of range

>>> x = {}

>>> x[42] = 'Foobar'

>>> x

{42: 'Foobar'}

First, I try to assign the string 'Foobar' to position 42 in an empty list—clearly impossible because that

position does not exist. To make this possible, I would have to initialize x with [None] \* 43 or something,

rather than simply []. The next attempt, however, works perfectly. Here I assign 'Foobar' to the key 42 of an

empty dictionary. You can see there’s no problem here. A new item is simply added to the dictionary, and

I’m in business.

Listing 4-1 shows the code for the telephone book example.

Listing 4-1. Dictionary Example

# A simple database

# A dictionary with person names as keys. Each person is represented as

# another dictionary with the keys 'phone' and 'addr' referring to their phone

# number and address, respectively.

people = {

'Alice': {

'phone': '2341',

'addr': 'Foo drive 23'

},

'Beth': {

'phone': '9102',

'addr': 'Bar street 42'

},

'Cecil': {

'phone': '3158',

'addr': 'Baz avenue 90'

}

}

# Descriptive labels for the phone number and address. These will be used

# when printing the output.

labels = {

'phone': 'phone number',

'addr': 'address'

}

name = input('Name: ')

# Are we looking for a phone number or an address?

request = input('Phone number (p) or address (a)? ')

# Use the correct key:

if request == 'p': key = 'phone'

if request == 'a': key = 'addr'

# Only try to print information if the name is a valid key in

# our dictionary:

if name in people: print("{}'s {} is {}.".format(name, labels[key], people[name][key]))

Here is a sample run of the program:

Name: Beth

Phone number (p) or address (a)? p

Beth's phone number is 9102.

String Formatting with Dictionaries

In Chapter 3, you saw how you could use string formatting to format values provided as individual (named

or unnamed) arguments to the format method. Sometimes, collecting a set of named values in the form

of a dictionary can make things easier. For example, the dictionary may contain all kinds of information,

and your format string will only pick out whatever it needs. You’ll have to specify that you’re supplying a

mapping, by using format\_map.

>>> phonebook

{'Beth': '9102', 'Alice': '2341', 'Cecil': '3258'}

>>> "Cecil's phone number is {Cecil}.".format\_map(phonebook)

"Cecil's phone number is 3258."

When using dictionaries like this, you may have any number of conversion specifiers, as long as all the given

keys are found in the dictionary. This sort of string formatting can be very useful in template systems (in this

case using HTML).

>>> template = '''<html>

... <head><title>{title}</title></head>

... <body>

... <h1>{title}</h1>

... <p>{text}</p>

... </body>'''

>>> data = {'title': 'My Home Page', 'text': 'Welcome to my home page!'}

>>> print(template.format\_map(data))

<html>

<head><title>My Home Page</title></head>

<body>

<h1>My Home Page</h1>

<p>Welcome to my home page!</p>

</body>

Dictionary Methods

Just like the other built-in types, dictionaries have methods. While these methods can be very useful, you

probably will not need them as often as the list and string methods. You might want to skim this section first

to get an idea of which methods are available and then come back later if you need to find out exactly how a

given method works.

clear

The clear method removes all items from the dictionary. This is an in-place operation (like list.sort), so it

returns nothing (or, rather, None).

>>> d = {}

>>> d['name'] = 'Gumby'

>>> d['age'] = 42

>>> d

{'age': 42, 'name': 'Gumby'}

>>> returned\_value = d.clear()

>>> d

{}

>>> print(returned\_value)

None

Why is this useful? Let’s consider two scenarios. Here’s the first one:

>>> x = {}

>>> y = x

>>> x['key'] = 'value'

>>> y

{'key': 'value'}

>>> x = {}

>>> x = {}

{'key': 'value'}

And here’s the second scenario:

>>> x = {}

>>> y = x

>>> x['key'] = 'value'

>>> y

{'key': 'value'}

>>> x.clear()

>>> y

{}

In both scenarios, x and y originally refer to the same dictionary. In the first scenario, I “blank out” x by

assigning a new, empty dictionary to it. That doesn’t affect y at all, which still refers to the original dictionary.

This may be the behavior you want, but if you really want to remove all the elements of the original

dictionary, you must use clear. As you can see in the second scenario, y is then also empty afterward.

copy

The copy method returns a new dictionary with the same key-value pairs (a shallow copy, since the values

themselves are the same, not copies).

>>> x = {'username': 'admin', 'machines': ['foo', 'bar', 'baz']}

>>> y = x.copy()

>>> y['username'] = 'mlh'

>>> y['machines'].remove('bar')

>>> y

{'username': 'mlh', 'machines': ['foo', 'baz']}

>>> x

{'username': 'admin', 'machines': ['foo', 'baz']}

As you can see, when you replace a value in the copy, the original is unaffected. However, if you modify a

value (in place, without replacing it), the original is changed as well because the same value is stored there

(like the 'machines' list in this example).

One way to avoid that problem is to make a deep copy, copying the values, any values they contain, and

so forth, as well. You accomplish this using the function deepcopy from the copy module.

>>> from copy import deepcopy

>>> d = {}

>>> d['names'] = ['Alfred', 'Bertrand']

>>> c = d.copy()

>>> dc = deepcopy(d)

>>> d['names'].append('Clive')

>>> c

{'names': ['Alfred', 'Bertrand', 'Clive']}

>>> dc

{'names': ['Alfred', 'Bertrand']}

fromkeys

The fromkeys method creates a new dictionary with the given keys, each with a default corresponding value

of None.

>>> {}.fromkeys(['name', 'age'])

{'age': None, 'name': None}

This example first constructs an empty dictionary and then calls the fromkeys method on that in order

to create another dictionary—a somewhat redundant strategy. Instead, you can call the method directly

on dict, which (as mentioned before) is the type of all dictionaries. (The concept of types and classes is

discussed more thoroughly in Chapter 7.)

>>> dict.fromkeys(['name', 'age'])

{'age': None, 'name': None}

If you don’t want to use None as the default value, you can supply your own default.

>>> dict.fromkeys(['name', 'age'], '(unknown)')

{'age': '(unknown)', 'name': '(unknown)'}

get

The get method is a forgiving way of accessing dictionary items. Ordinarily, when you try to access an item

that is not present in the dictionary, things go very wrong.

>>> d = {}

>>> print(d['name'])

Traceback (most recent call last):

File "<stdin>", line 1, in ?

KeyError: 'name'

That isn’t the case with get.

>>> print(d.get('name'))

None

As you can see, when you use get to access a nonexistent key, there is no exception. Instead, you get the

value None. You may supply your own “default” value, which is then used instead of None.

>>> d.get('name', 'N/A')

'N/A'

If the key is there, get works like ordinary dictionary lookup.

>>> d['name'] = 'Eric'

>>> d.get('name')

'Eric'

Listing 4-2 shows a modified version of the program from Listing 4-1, which uses the get method to access

the “database” entries.

Listing 4-2. Dictionary Method Example

# A simple database using get()

# Insert database (people) from Listing 4-1 here.

labels = {

'phone': 'phone number',

'addr': 'address'

}

name = input('Name: ')

# Are we looking for a phone number or an address?

request = input('Phone number (p) or address (a)? ')

# Use the correct key:

key = request # In case the request is neither 'p' nor 'a'

if request == 'p': key = 'phone'

if request == 'a': key = 'addr'

# Use get to provide default values:

person = people.get(name, {})

label = labels.get(key, key)

result = person.get(key, 'not available')

print("{}'s {} is {}.".format(name, label, result))

An example run of this program follows. Notice how the added flexibility of get allows the program to give

a useful response, even though the user enters values we weren’t prepared for.

Name: Gumby

Phone number (p) or address (a)? batting average

Gumby's batting average is not available.

items

The items method returns all the items of the dictionary as a list of items in which each item is of the form

(key, value). The items are not returned in any particular order.

>>> d = {'title': 'Python Web Site', 'url': 'http://www.python.org', 'spam': 0}

>>> d.items()

dict\_items([('url', 'http://www.python.org'), ('spam', 0), ('title', 'Python Web Site')])

The return value is of a special type called a dictionary view. Dictionary views can be used for iteration

(see Chapter 5 for more on that). In addition, you can determine their length and check for membership.

>>> it = d.items()

>>> len(it)

3

>>> ('spam', 0) in it

True

A useful thing about views is that they don’t copy anything; they always reflect the underlying dictionary,

even if you modify it.

>>> d['spam'] = 1

>>> ('spam', 0) in it

False

>>> d['spam'] = 0

>>> ('spam', 0) in it

True

If, however, you’d rather copy the items into a list (which is what happened when you used items in older

versions of Python), you can always do that yourself.

>>> list(d.items())

[('spam', 0), ('title', 'Python Web Site'), ('url', 'http://www.python.org')]

keys

The keys method returns a dictionary view of the keys in the dictionary.

pop

The pop method can be used to get the value corresponding to a given key and then to remove the key-value

pair from the dictionary.

>>> d = {'x': 1, 'y': 2}

>>> d.pop('x')

1

>>> d

{'y': 2}

popitem

The popitem method is similar to list.pop, which pops off the last element of a list. Unlike list.pop,

however, popitem pops off an arbitrary item because dictionaries don’t have a “last element” or any order

whatsoever. This may be very useful if you want to remove and process the items one by one in an efficient

way (without retrieving a list of the keys first).

>>> d = {'url': 'http://www.python.org', 'spam': 0, 'title': 'Python Web Site'}

>>> d.popitem()

('url', 'http://www.python.org')

>>> d

{'spam': 0, 'title': 'Python Web Site'}

Although popitem is similar to the list method pop, there is no dictionary equivalent of append (which adds

an element to the end of a list). Because dictionaries have no order, such a method wouldn’t make any sense.

■■Tip If you want the popitem method to follow a predictable ordering, take a look at the OrderedDict class

from the collections module.

setdefault

The setdefault method is somewhat similar to get, in that it retrieves a value associated with a given key. In

addition to the get functionality, setdefault sets the value corresponding to the given key if it is not already

in the dictionary.

>>> d = {}

>>> d.setdefault('name', 'N/A')

'N/A'

>>> d

{'name': 'N/A'}

>>> d['name'] = 'Gumby'

>>> d.setdefault('name', 'N/A')

'Gumby'

>>> d

{'name': 'Gumby'}

As you can see, when the key is missing, setdefault returns the default and updates the dictionary

accordingly. If the key is present, its value is returned, and the dictionary is left unchanged. The default is

optional, as with get; if it is left out, None is used.

>>> d = {}

>>> print(d.setdefault('name'))

None

>>> d

{'name': None}

■■Tip

If you want a global default for the entire dictionary, check out the defaultdict class from the

collections module.

update

The update method updates one dictionary with the items of another.

>>> d = {

... 'title': 'Python Web Site',

... 'url': 'http://www.python.org',

... 'changed': 'Mar 14 22:09:15 MET 2016'

... }

>>> x = {'title': 'Python Language Website'}

>>> d.update(x)

>>> d

{'url': 'http://www.python.org', 'changed':

'Mar 14 22:09:15 MET 2016', 'title': 'Python Language Website'}

The items in the supplied dictionary are added to the old one, supplanting any items there with the

same keys.

The update method can be called in the same way as the dict function (or type constructor), as

discussed earlier in this chapter. This means that update can be called with a mapping, a sequence (or other

iterable object) of (key, value) pairs, or keyword arguments.

values

The values method returns a dictionary view of the values in the dictionary. Unlike keys, the view returned

by values may contain duplicates.

>>> d = {}

>>> d[1] = 1

>>> d[2] = 2

>>> d[3] = 3

>>> d[4] = 1

>>> d.values()

dict\_values([1, 2, 3, 1])

A Quick Summary

In this chapter, you learned about the following:

Mappings: A mapping enables you to label its elements with any immutable

object, the most usual types being strings and tuples. The only built-in mapping

type in Python is the dictionary.

String formatting with dictionaries: You can apply the string formatting

operation to dictionaries by using format\_map, rather than using named

arguments with format.

Dictionary methods: Dictionaries have quite a few methods, which are called in

the same way as list and string methods.

New Functions in This Chapter

FunctionDescription

dict(seq)Creates dictionary from (key, value) pairs (or a mapping or keyword arguments)

What Now?

You now know a lot about Python’s basic data types and how to use them to form expressions. As you may

remember from Chapter 1, computer programs have another important ingredient—statements. They’re

covered in detail in the next chapter.