Lesson 27. Dictionaries as maps between objects

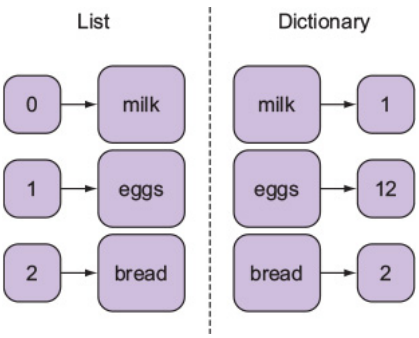
After reading lesson 27, you'll be able to

- Understand what a dictionary object data type is
- Add to, remove from, and look up objects in dictionaries
- Understand when to use a dictionary object
- Understand the difference between a dictionary and a list

In the previous lesson, you learned about lists as collections of data with elements at a certain position in the list. Lists are useful when you want to store one group of objects; you saw that you could store a group of names or a group of numbers. But in real life, you often have pairs of data: a word and its meaning, a word and a list of synonyms, a person and their phone number, a movie and its rating, a song and its artist, and many others.

Figure 27.1 takes the grocery list metaphor from the previous lesson and shows you one way to apply it to dictionaries. In a list, your grocery items are enumerated; the first item is on the first line, and so on. You can think of a list as mapping the numbers 0, 1, 2, and so on, in that order, to each item in your list. With a dictionary, you get additional flexibility in what you can map, and to what. In figure 27.1, the grocery dictionary now maps an item to its quantity.

Figure 27.1. A list puts the first item at position 0, the second item at position 1, and the third at position 2. A dictionary doesn't have positions, but rather maps one object to another; here it maps a grocery item to its quantity.



You can think of a list as a structure that maps an integer (index 0, 1, 2, 3 ...) to an object; a list can be indexed only by an integer. A dictionary is a data structure that can map any object, not just an integer, to any other object; indexing using any object is more useful in situations when you have pairs of data. Like lists, dictionaries are mutable so that when you make a change to a dictionary object, the object itself changes without having to make a copy of it.

Consider this

A word dictionary maps words from one language to their equivalent in another language. For each of the following scenarios, are you able to map one thing to another?

- Your friends and their phone numbers
- All the movies you've seen
- The number of times each word occurs in your favorite song
- Each coffee shop in the area and its Wi-Fi availability
- All the names of the paints available at the hardware store
- Your coworkers and their arrival and leave times

Answer:

- No
- Yes
- Yes
- No
- Yes

27.1. CREATING DICTIONARIES, KEYS, AND VALUES

Many programming languages have a way to map objects to each other or to look up one object using another. In Python, such an object is called a *dictionary* with the object type `dict`.

A dictionary maps one object to another; we can also say that you look up one object by using another object. If you think of a traditional word dictionary, you look up a word to find its meaning. In programming, the item you look up (in a traditional dictionary, the *word*) is called the *key*, and what the item lookup returns (in a traditional dictionary, the *meaning*) is called the *value*. In programming, a dictionary stores entries, and each entry is a key-value pair. You use one object (the key) to look up another object (the value).

You create an empty Python dictionary with curly braces:

```
grocery = {}
```

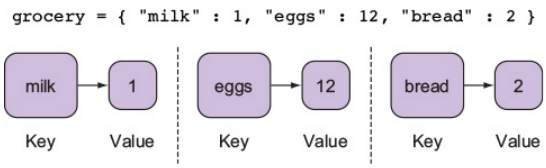
This command creates an empty dictionary with no entries and binds the dictionary object to a variable named `grocery`.

You can also create a dictionary with items already in it. In your grocery list, you map a grocery item to its quantity. In other words, the keys to `grocery` will be strings representing the grocery items, and the values to `grocery` will be integers representing the quantity:

```
grocery = {"milk": 1, "eggs": 12, "bread": 2}
```

This line creates a dictionary with three items, as shown in figure 27.2. Each item in a dictionary is separated by a comma. The keys and values for an item are separated by a colon. The key is to the left of the colon, and the value for that key is to the right of the colon.

Figure 27.2. A dictionary initialized with three entries. Entries are separated by commas. Each entry has a key to the left of a colon, and the key's corresponding value to the right of the colon.



Both keys and values to a dictionary are single objects. A dictionary entry can't have more than one object as its key or more than one object as its value. If you'd like to store more than one object as the value, you could store all the objects inside a tuple, because a tuple is one object. For example, your grocery list could have the string key `"eggs"` and the tuple value `(1, "carton")` or `(12, "individual")`. Notice that the values in both situations are one Python object, a tuple.

Quick check 27.1

For each of the following situations, write a line of code that creates a dictionary and binds it to a variable with an appropriate name. For each, also indicate the keys and values:

1

Empty dictionary of employee names and their phone numbers and addresses

2

Empty dictionary of cities and the number of inches of snow each city got in 1990 and 2000

3

Dictionary of items in a house and their value: a TV worth $2,000 and a sofa worth $1,500

Quick check 27.2

For each of the following, how many entries are in the dictionary? What's the type of the keys, and what's the type of the values?

1

```
d = {1:-1, 2:-2, 3:-3}
```

2

```
d = {"1":1, "2":2, "3":3}
```

3

```
d = {2:[0,2], 5:[1,1,1,1,1], 3:[2,1,0]}
```

27.2. ADDING KEY-VALUE PAIRS TO A DICTIONARY

Empty dictionaries or dictionaries with a fixed number of entries aren't useful. You'll want to add more entries to store more information. To add a new key-value pair, you use square brackets, much as with lists:

```
d[k] = v
```

This command adds the key `k` and its associated value `v` into the dictionary `d`. If you try to add to the same key again, the previous value associated with that key will be overwritten.

At any point, you can use the `len()` function to tell you the number of entries in the dictionary. The following listing adds items to a dictionary.

Listing 27.1. Adding pairs to a dictionary

```
legs = {}                          1
legs["human"] = 2                  2
legs["cat"] = 4                    3
legs["snake"] = 0                  4
print(len(legs))                5
legs["cat"] = 3                 6
print(len(legs))                7
print(legs)                     8
```

- ***1*** **Empty dictionary**
- ***2*** **Adds key "human" with value 2**
- ***3*** **Adds key "cat" with value 4**
- ***4*** **Adds key "snake" with value 0**
- ***5*** **Prints 3 because there are three entries in the dictionary**
- ***6*** **Changes key "cat" to value 3**
- ***7*** **Prints 3 because you modified only the entry for "cat"**
- ***8*** **Prints {'human': 2, 'snake': 0, 'cat': 3}**

The preceding code shows the output using Python 3.5. If you use a different version of Python, you may see a different order in the dictionary. Using the output from Python 3.5, you may have noticed that the items you added to the dictionary were a human, then a cat, and then a snake. But when you printed the dictionary in listing 27.1, the dictionary printed a different order. This is normal behavior with dictionaries, and section 27.4.1 discusses this further.

27.2.1. Short diversion into restrictions on keys

When you try to insert an object key into a dictionary that already contains that key, the value for the existing key is overwritten. This leads to an interesting point about the kinds of objects you can store as keys in a dictionary.

You can't have the same key multiple times in a dictionary; if you did, Python wouldn't know which key you're referring to when you want to retrieve a value. For example, if you have the word `box` as a key that maps to `container`, and the word `box` that also maps to `fight`, then which definition do you give to someone who wants the definition of `box`? The first one you find? The last one? Both? The answer isn't so clear. Instead of dealing with this situation, Python guarantees that all keys in a dictionary are unique objects.

How and why does the Python language make this guarantee? Python enforces that a key is an immutable object. This decision is because of the way that Python implements the dictionary object.

method is used so that when you want to look up a value, you can quickly retrieve the value by running the formula instead of iterating through all the keys to find the one you want. Because the result of the hash function should be the same when inserting or when looking up an item, the location where the value is stored is fixed. Using an immutable object, you'll get the same location value because the immutable object's value doesn't change. But if you used a mutable object, it's possible (and likely) that the hash function would give a different location value when applied to the mutated key value, as opposed to the original key value.

Quick check 27.3

Q1:

What's the value of the dictionary after each line is executed?

```
city_pop = {}
city_pop["LA"] = 3884
city_pop["NYC"] = 8406
city_pop["SF"] = 837
city_pop["LA"] = 4031
```

27.3. REMOVING KEY-VALUE PAIRS FROM A DICTIONARY

As with lists, you can remove items after you put them into a dictionary by using the `pop()` operation. The command `d.pop(k)` will remove the key-value entry in the dictionary `d` corresponding to the key `k`. This operation is like a function and returns the value from the dictionary associated with the key `k`. After the pop operation in the next listing, the dictionary `household` will have the value `{"person":4, "cat":2, "dog":1}`.

Listing 27.2. Removing pairs from a dictionary

```
household = {"person":4, "cat":2, "dog":1, "fish":2}
removed = household.pop("fish")
print(removed)
```

- ***1*** **Fills a dictionary**
- ***2*** **Removes entry whose key is "fish" and saves the value associated with the key "fish" in a variable**
- ***3*** **Prints value of removed entry**

Quick check 27.4

Q1:

What's printed by the following code? If there's an error, write error:

```
constants = {"pi":3.14, "e":2.72, "pyth":1.41, "g
print(constants.pop("pi"))
print(constants.pop("pyth"))
print(constants.pop("i"))
```

27.4. GETTING ALL THE KEYS AND VALUES IN A DICTIONARY

Python has two operations that allow you to get all the keys in the dictionary and all the values in the dictionary. This is useful if you need to look through all the pairs to find entries that match certain criteria. For example, if you have a dictionary that maps a song name to its rating, you may want to retrieve all the pairs, and keep only the ones whose rating is 4 or 5.

If a dictionary named `songs` contains pairs of songs and ratings, you can use `songs.keys()` to get all the keys in the dictionary. The following code prints `dict_keys(['believe', 'roar', 'let it be'])`:

```
songs = {"believe": 3, "roar": 5, "let it be": 4}
print(songs.keys())
```

The expression `dict_keys(['believe', 'roar', 'let it be'])` is a Python object that contains all the keys in the dictionary.

You can iterate over the returned keys directly by using a `for` loop, as in this line:

```
for one_song in songs.keys():
```

Alternatively, you can save the keys in a list by casting the returned keys into a list via this command:

```
all_songs = list(songs.keys())
```

Similarly, the command `songs.values()` gives you all the values in the dictionary `songs`. You can either iterate over them directly or cast them into a list if you need to use them later in your code. It's most often useful to iterate over the keys in a dictionary because after you know a key, you can always look up the value corresponding to that key.

Let's look at a different example. Suppose you have data for the following students in your class:

```
Name        Quiz 1 Grade   Quiz 2 Grade
Chris       100            70
Angela      90             100
Bruce       80             40
Stacey      70             70
```

Listing 27.3 shows an example of how to use dictionary commands to keep track of students and their grades in your class. First, you create a dictionary that maps student names to their grades on exams. Assuming each student took two exams, the value in the dictionary for each student will be a list containing two elements. Using the dictionary, you can print all the names of the students in the class by iterating through all the keys, and you can also iterate through all the values and print all the averages for the quizzes. Lastly, you can even modify the values for each key by adding to the end of the list the average of the two quizzes.

Listing 27.3. Keeping track of student grades by using a dictionary

```
grades = {}                                          1
grades["Chris"] = [100, 70]                          1
grades["Angela"] = [90, 100]                         1
grades["Bruce"] = [80, 40]                           1
grades["Stacey"] = [70, 70]                          1

for student in grades.keys():                        2
    print(student)                                   2

for quizzes in grades.values():                      3
    print(sum(quizzes)/2)                            2

for student in grades.keys():                        4
    scores = grades[student]                         5
    grades[student].append(sum(scores)/2)            6
print(grades)                                        7
```

- ***1* Sets up the dictionary mapping a string to a list of the two quiz scores**
- ***2* Iterates through keys and prints them**
- ***3* Iterates through values and prints their average**
- ***4* Iterates through all keys**
- ***5* Takes scores of each student and assigns them to the scores variable for average calculation in the next line**
- ***6* Takes the average of the elements and appends it to the end of the list**
- ***7* Prints {'Bruce': [80, 40, 60.0], 'Stacey': [70, 70, 70.0], 'Angela': [90, 100, 95.0], 'Chris': [100, 70, 85.0]}**

Quick check 27.5

Q1:

You have the following lines of code that perform operations on an employee database by incrementing everyone's age by 1. What does the code print?

```
employees = {"John": 34, "Mary": 24, "Erin": 50}
for em in employees.keys():
    employees[em] += 1
for em in employees.keys():
    print(employees[em])
```

27.4.1. No ordering to dictionary pairs

listing 27.3, you may notice something odd. The output from printing all the keys is as follows:

```
Bruce
Stacey
Angela
Chris
```

But when you added items to the dictionary, you added Chris, then Angela, then Bruce, then Stacey. These orders don't seem to match. Unlike lists, a Python dictionary forgets the order in which items were added to the dictionary. When you ask for keys or values, you have no guarantee about the order in which they're returned. You can see this by typing in the following code that checks for equality between two dictionaries and then between two lists:

```
print({"Angela": 70, "Bruce": 50} == {"Bruce": 50, "A
print(["Angela", "Bruce"] == ["Bruce", "Angela"])
```

The two dictionaries are equal, even though the order that the entries are put in aren't the same. In contrast, a list of the two names must be in the same order to be considered equal.

27.5. WHY SHOULD YOU USE A DICTIONARY?

By now, it should be clear that dictionaries can be fairly useful objects because they map objects (keys) to other objects (values), and you can later look up values given a key. Dictionaries have two common uses: keeping count of the number of times something occurs, and using dictionaries to map items to functions.

27.5.1. Keeping count with frequency dictionaries

Possibly one of the most common uses of dictionaries is to keep track of the quantity of a certain item. For example, if you're writing a Scrabble game, you may use a dictionary to keep track of the quantity of each letter in your hand. If you have a text document, you may want to keep track of the number of times you use each word. In listing 27.4, you'll build a frequency dictionary that maps a word to the number of times it occurs in a song. The code takes a string and makes a list of words by splitting on the space. With an initially empty dictionary, you go through all the words in the list and do one of two things:

- If you haven't added the word to the dictionary yet, add it with a count of 1.
- If you've already added the word to the dictionary, increase its count by 1.

Listing 27.4. Building a frequency dictionary

```
lyrics = "Happy birthday to you Happy birthday to you
➥ Happy birthday to you"                          1
counts = {}                                        2

words = lyrics.split(" ")                          3
for w in words:                                    4
    w = w.lower()                                  5
    if w not in counts:
        counts[w] = 1                              6
    else:
        counts[w] += 1                             7

print(counts)                                      8
```

- ***1*** **String with song lyrics**
- ***2*** **Empty frequency dictionary**
- ***3*** **Gets a list of all words in the string by splitting the string on the space character**
- ***4*** **Iterates through each word in the list from the preceding line**
- ***5*** **Converts to lowercase**
- ***6*** **Word isn't in the dictionary yet, so add it as the key and set its value to 1**
- ***7*** **Word is already in the dictionary, so increment its count by 1**
- ***8*** **Prints {'happy': 4, 'to': 3, 'dear': 1, 'you': 3, 'birthday': 4}**

A frequency dictionary is a useful application of Python dictionaries, and you'll write a function to build a frequency dictionary in the capstone project in lesson 29.

27.5.2. Building unconventional dictionaries

need to map two items and access them later, a dictionary should be the first thing you try to use. But there are some less obvious scenarios for using a dictionary. One use is to map common names to functions. In listing 27.5, you define three functions that, given an input variable, find the area of three common shapes: a square, circle, and equilateral triangle.

You can build a dictionary that maps a string to the function itself, referenced by the function name. When you look up each string, you'll get back the function object. Then, using the function object, you can call it using a parameter. In listing 27.5, when you access the dictionary using `"sq"` in the line `print(areas["sq"](n))`, the value retrieved by `areas["sq"]` is the function named `square`. The function is then called on the number `n = 2` when you use `areas["sq"](n)`.

Listing 27.5. Dictionaries and functions

```
def square(x):
    return x*x

def circle(r):
    return 3.14*r*r

def equilateraltriangle(s):
    return (s*s)*(3**0.5)/4

areas = {"sq": square, "ci": circle, "eqtri": equilat

n = 2
print(areas["sq"](n))
print(areas["ci"](n))
print(areas["eqtri"](n))
```

- ***1*** **Known function to calculate area of a square**
- ***2*** **Known function to calculate area of a circle**
- ***3*** **Known function to calculate area of an equilateral triangle**
- ***4*** **Dictionary maps string to function**
- ***5*** **Calls function mapped in dictionary by key "sq" on n where n is 2**
- ***6*** **Calls function mapped in dictionary by key "ci" on n where n is 2**
- ***7*** **Calls function mapped in dictionary by key "eqtri" on n where n is 2**

SUMMARY

In this lesson, my objective was to teach you a new data type, the Python dictionary. A dictionary maps one object to another. Like lists, dictionaries are mutable objects in which you can add to, remove from, and change elements. Unlike lists, dictionaries don't have an order, and they allow only certain object types to be the keys. Here are the major takeaways:

- Dictionaries are mutable.
- Dictionary keys must be immutable objects.
- Dictionary values can be mutable or immutable.
- A dictionary doesn't have an order.

Let's see if you got this...

Q27.1

Write a program that uses dictionaries to accomplish the following task. Given a dictionary of song names (strings) mapped to ratings (integers), print the song names of all songs that are rated exactly 5.

Q27.2

Write a function named `replace`. It takes in one dictionary, `d`, and two values, `v` and `e`. The function doesn't return anything. It mutates `d` such that all the values `v` in `d` are replaced with `e`. For example,

- `replace({1:2, 3:4, 4:2}, 2, 7)` mutates `d` to `{1: 7, 3: 4, 4: 7}`.
- `replace({1:2, 3:1, 4:2}, 1, 2)` mutates `d` to `{1: 2, 3: 2, 4: 2}`.

Q27.3

Write a function named `invert`. It takes in one dictionary, `d`. The function returns a new dictionary, `d_inv`. The keys in `d_inv` are the unique values in `d`. The value corresponding to a key in `d_inv` is a list. The list contains all the keys in `d` that mapped to the same value in `d`. For example,

- `invert({1:2, 3:4, 5:6})` returns `{2: [1], 4: [3], 6:`

- `invert({1:2, 2:1, 3:3})` returns `{1: [2], 2: [1], 3: [3]}`.
- `invert({1:1, 3:1, 5:1})` returns `{1: [1, 3, 5]}`.

Recommended / Playlists / History / Topics / Settings / Get the App / Sign Out

PREV
Lesson 26. Advanced operations with lists

NEXT
Lesson 28. Aliasing and copying lists and dictionaries