Dictionaries 6

There are many parallels between lists and dictionaries. Like lists, dictionaries allow several, even many, values to be stored in one variable. Each element in a list has a unique integer index associated with it, and these indices must be integers that increase sequentially from zero. Similarly, each value in a dictionary has a unique *key* associated with it, but a dictionary's keys are more flexible than a list's indices. A dictionary's keys can be integers. They can also be floating-point numbers or strings. When the keys are numeric they do not have to start from zero, nor do they have to be sequential. When the keys are strings they can be any combination of characters, including the empty string. All of the keys in a dictionary must be distinct just as all of the indices in a list are distinct.

Every key in a dictionary must have a *value* associated with it. The value associated with a key can be an integer, a floating-point number, a string or a Boolean value. It can also be a list, or even another dictionary. A dictionary key and it's corresponding value are often referred to as a *key-value pair*. While the keys in a dictionary must be distinct there is no parallel restriction on the values. Consequently, the same value can be associated with multiple keys.

Starting in Python 3.7, the key-value pairs in a dictionary are always stored in the order in which they were added to the dictionary.[1] Each time a new key-value pair is added to the dictionary it is added to the end of the existing collection. There is no mechanism for inserting a key-value pair in the middle of an existing dictionary. Removing a key-value pair from the dictionary does not change the order of the remaining key-value pairs in the dictionary.

A variable that holds a dictionary is created using an assignment statement. The empty dictionary, which does not contain any key-value pairs, is denoted by `{}` (an open brace immediately followed by a close brace). A non-empty dictionary can be created by including a comma separated collection of key-value pairs inside the

[1]The order in which the key-value pairs were stored was not guaranteed to be the order in which they were added to the dictionary in earlier versions of Python.

B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_6

braces. A colon is used to separate the key from its value in each key-value pair. For example, the following program creates a dictionary with three key-value pairs where the keys are strings and the values are floating-point numbers. Each key-value pair associates the name of a common mathematical constant to its value. Then all of the key-value pairs are displayed by calling the `print` function.

```
constants = {"pi": 3.14, "e": 2.71, "root 2": 1.41}
print(constants)
```

6.1 Accessing, Modifying and Adding Values

Accessing a value in a dictionary is similar to accessing a value in a list. When the index of a value in a list is known, we can use the name of the list and the index enclosed in square brackets to access the value at that location. Similarly, when the key associated with a value in a dictionary is known, we can use the name of the dictionary and the key enclosed in square brackets to access the value associated with that key.

Modifying an existing value in a dictionary and adding a new key-value pair to a dictionary are both performed using assignment statements. The name of the dictionary, along with the key enclosed in square brackets, is placed to the left of the assignment operator, and the value to associate with the key is placed to the right of the assignment operator. If the key is already present in the dictionary then the assignment statement will replace the key's current value with the value to the right of the assignment operator. If the key is not already present in the dictionary then a new key-value pair is added to it. These operations are demonstrated in the following program.

```
# Create a new dictionary with 2 key-value pairs
results = {"pass": 0, "fail": 0}

# Add a new key-value pair to the dictionary
results["withdrawal"] = 1

# Update two values in the dictionary
results["pass"] = 3
results["fail"] = results["fail"] + 1

# Display the values associated with fail, pass and withdrawal respectively
print(results["fail"])
print(results["pass"])
print(results["withdrawal"])
```

When this program executes it creates a dictionary named `results` that initially has two keys: `pass` and `fail`. The value associated with each key is 0. A third key, `withdrawal`, is added to the dictionary with the value 1 using an assignment statement. Then the value associated with `pass` is updated to 3 using a second assignment statement. The line that follows reads the current value associated with `fail`, which is 0, adds 1 to it, and then stores this new value back into the dictionary, replacing the previous value. When the values are printed 1 (the value currently associated with fail) is displayed on the first line, 3 (the value currently associated with pass) is displayed on the second line, and 1 (the value currently associated with withdrawal) is displayed on the third line.

6.2 Removing a Key-Value Pair

A key-value pair is removed from a dictionary using the pop method. One argument, which is the key to remove, must be supplied when the method is called. When the method executes it removes both the key and the value associated with it from the dictionary. Unlike a list, it is not possible to pop the last key-value pair out of a dictionary by calling pop without any arguments.

The pop method returns the value associated with the key that is removed from the dictionary. This value can be stored into a variable using an assignment statement, or it can be used anywhere else that a value is needed, such as passing it as an argument to another function or method call, or as part of an arithmetic expression.

6.3 Additional Dictionary Operations

Some programs add key-value pairs to dictionaries where the key or the value were read from the user. Once all of the key-value pairs have been stored in the dictionary it might be necessary to determine how many there are, whether a particular key is present in the dictionary, or whether a particular value is present in the dictionary. Python provides functions, methods and operators that allow us to perform these tasks.

The len function, which we previously used to determine the number of elements in a list, can also be used to determine how many key-value pairs are in a dictionary. The dictionary is passed as the only argument to the function, and the number of key-value pairs is returned as the function's result. The len function returns 0 if the dictionary passed as an argument is empty.

The in operator can be used to determine whether or not a particular key or value is present in a dictionary. When searching for a key, the key appears to the left of the in operator and a dictionary appears to its right. The operator evaluates to True if the key is present in the dictionary. Otherwise it evaluates to False. The result returned by the in operator can be used anywhere that a Boolean value is needed, including in the condition of an if statement or while loop.

The in operator is used together with the values method to determine whether or not a value is present in a dictionary. The value being searched for appears to the left of the in operator and a dictionary, with the values method applied to it, appears to its right. For example, the following code segment determines whether or not any of the values in dictionary d are equal to the value that is currently stored in variable x.

```
if x in d.values():
  print("At least one of the values in d is", x)
else:
  print("None of the values in d are", x)
```

6.4 Loops and Dictionaries

A `for` loop can be used to iterate over all of the keys in a dictionary, as shown below. A different key from the dictionary is stored into the `for` loop's variable, `k`, each time the loop body executes.

```
# Create a dictionary
constants = {"pi": 3.14, "e": 2.71, "root 2": 1.41}

# Print all of the keys and values with nice formatting
for k in constants:
  print("The value associated with", k, "is", constants[k])
```

When this program executes it begins by creating a new dictionary that contains three key-value pairs. Then the `for` loop iterates over the keys in the dictionary. The first key in the dictionary, which is `pi`, is stored into `k`, and the body of the loop executes. It prints out a meaningful message that includes both `pi` and its value, which is `3.14`. Then control returns to the top of the loop and `e` is stored into `k`. The loop body executes for a second time and displays a message indicating that the value of `e` is `2.71`. Finally, the loop executes for a third time with `k` equal to `root 2` and the final message is displayed.

A `for` loop can also be used to iterate over the values in a dictionary (instead of the keys). This is accomplished by applying the `values` method, which does not take an arguments, to a dictionary to create the collection of values used by the `for` loop. For example, the following program computes the sum of all of the values in a dictionary. When it executes, `constants.values()` will be a collection that includes 3.14, 2.71 and 1.41. Each of these values is stored in `v` as the `for` loop runs, and this allows the total to be computed without using any of the dictionary's keys.

```
# Create a dictionary
constants = {"pi": 3.14, "e": 2.71, "root 2": 1.41}

# Compute the sum of all the value values in the dictionary
total = 0
for v in constants.values():
  total = total + v

# Display the total
print("The total is", total)
```

Some problems involving dictionaries are better solved with `while` loops than `for` loops. For example, the following program uses a `while` loop to read strings from the user until 5 unique values have been entered. Then all of the strings are displayed with their counts.

```
# Count how many times each string is entered by the user
counts = {}

# Loop until 5 distinct strings have been entered
while len(counts) < 5:
  s = input("Enter a string: ")

  # If s is already a key in the dictionary then increase its count by 1. Otherwise add s to the
  # dictionary with a count of 1.
  if s in counts:
    counts[s] = counts[s] + 1
  else:
    counts[s] = 1

# Displays all of the strings and their counts
for k in counts:
  print(k, "occurred", counts[k], "times")
```

When this program executes it begins by creating an empty dictionary. Then the `while` loop condition is evaluated. It determines how many key-value pairs are in the dictionary using the `len` function. Since the number of key-value pairs is initially 0, the condition evaluates to `True` and the loop body executes.

Each time the loop body executes a string is read from the user. Then the `in` operator is used to determine whether or not the string is already a key in the dictionary. If so, the count associated with the key is increased by one. Otherwise the string is added to the dictionary as a new key with a value of 1. The loop continues executing until the dictionary contains 5 key-value pairs. Once this occurs, all of the strings that were entered by the user are displayed, along with their associated values.

6.5 Dictionaries as Arguments and Return Values

Dictionaries can be passed as arguments to functions, just like values of other types. As with lists, a change made to a parameter variable that contains a dictionary can modify both the parameter variable and the argument passed to the function. For example, inserting or deleting a key-value pair will modify both the parameter variable and the argument, as will modifying the value associated with one key in the dictionary using an assignment statement. However an assignment to the entire dictionary (where only the name of the dictionary appears to the left of the assignment operator) only impacts the parameter variable. It does not modify the argument passed to the function. As with other types, dictionaries are returned from a function using the `return` keyword.

6.6 Exercises

While many of the exercises in this chapter can be solved with lists or `if` statements, most (or even all) of them have solutions that are well suited to dictionaries. As a result, you should use dictionaries to solve all of these exercises instead of (or in addition to) using the Python features that you have been introduced to previously.

Exercise 136: Reverse Lookup

(*Solved, 45 Lines*)

Write a function named `reverseLookup` that finds all of the keys in a dictionary that map to a specific value. The function will take the dictionary and the value to search for as its only parameters. It will return a (possibly empty) list of keys from the dictionary that map to the provided value.

Include a main program that demonstrates the `reverseLookup` function as part of your solution to this exercise. Your program should create a dictionary and then show that the `reverseLookup` function works correctly when it returns multiple keys, a single key, and no keys. Ensure that your main program only runs when the file containing your solution to this exercise has not been imported into another program.

Exercise 137: Two Dice Simulation

(*Solved, 43 Lines*)

In this exercise you will simulate 1,000 rolls of two dice. Begin by writing a function that simulates rolling a pair of six-sided dice. Your function will not take any parameters. It will return the total that was rolled on two dice as its only result.

Write a main program that uses your function to simulate rolling two six-sided dice 1,000 times. As your program runs, it should count the number of times that each total occurs. Then it should display a table that summarizes this data. Express the frequency for each total as a percentage of the number of rolls performed. Your program should also display the percentage expected by probability theory for each total. Sample output is shown below.

```
Total   Simulated   Expected
          Percent    Percent
    2        2.90       2.78
    3        6.90       5.56
    4        9.40       8.33
    5       11.90      11.11
    6       14.20      13.89
    7       14.20      16.67
    8       15.00      13.89
    9       10.50      11.11
   10        7.90       8.33
   11        4.50       5.56
   12        2.60       2.78
```

Exercise 138: Text Messaging

(*21 Lines*)

On some basic cell phones, text messages can be sent using the numeric keypad. Because each key has multiple letters associated with it, multiple key presses are needed for most letters. Pressing the number once generates the first character listed for that key. Pressing the number 2, 3, 4 or 5 times generates the second, third, fourth or fifth character.

Key	Symbols
1	. , ? ! :
2	A B C
3	D E F
4	G H I
5	J K L
6	M N O
7	P Q R S
8	T U V
9	W X Y Z
0	*space*

Write a program that displays the key presses needed for a message entered by the user. Construct a dictionary that maps from each letter or symbol to the key presses needed to generate it. Then use the dictionary to create and display the presses needed for the user's message. For example, if the user enters `Hello, World!` then your program should output `4433555555666110966677755531111`. Ensure that your program handles both uppercase and lowercase letters. Ignore any characters that aren't listed in the table above such as semicolons and parentheses.

Exercise 139: Morse Code

(*15 Lines*)

Morse code is an encoding scheme that uses dashes and dots to represent digits and letters. In this exercise, you will write a program that uses a dictionary to store the mapping from these symbols to Morse code. Use a period to represent a dot, and a hyphen to represent a dash. The mapping from characters to dashes and dots is shown in Table 6.1.

Your program should read a message from the user. Then it should translate all of the letters and digits in the message to Morse code, leaving a space between each sequence of dashes and dots. Your program should ignore any characters that are not listed in the previous table. The Morse code for `Hello, World!` is shown below:

```
.... . .-.. .-.. --- .-- --- .-. .-.. -..
```

Table 6.1 Morse code for letters and numerals

Character	Code	Character	Code	Character	Code	Character	Code
A	. −	J	. − − −	S	. . .	1	. − − − −
B	− . . .	K	− . −	T	−	2	. . − − −
C	− . − .	L	. − . .	U	. . −	3	. . . − −
D	− . .	M	− −	V	. . . −	4	. . . . −
E	.	N	− .	W	. − −	5	. . . . .
F	. . − .	O	− − −	X	− . . −	6	− . . . .
G	− − .	P	. − − .	Y	− . − −	7	− − . . .
H	. . . .	Q	− − . −	Z	− − . .	8	− − − . .
I	. .	R	. − .	0	− − − − −	9	− − − − .

Morse code was originally developed in the nineteenth century for use over telegraph wires. It is still used today, more than 160 years after it was first created.

Exercise 140: Postal Codes

(*24 Lines*)

The first, third and fifth characters in a Canadian postal code are letters while the second, fourth and sixth characters are digits. The province or territory in which an address resides can be determined from the first character of its postal code, as shown in the following table. No valid postal codes currently begin with D, F, I, O, Q, U, W, or Z.

Province / Territory	First Character(s)
Newfoundland	A
Nova Scotia	B
Prince Edward Island	C
New Brunswick	E
Quebec	G, H and J
Ontario	K, L, M, N and P
Manitoba	R
Saskatchewan	S
Alberta	T
British Columbia	V
Nunavut	X
Northwest Territories	X
Yukon	Y

The second character in a postal code identifies whether the address is rural or urban. If that character is a 0 then the address is rural. Otherwise it is urban.

Create a program that reads a postal code from the user and displays the province or territory associated with it, along with whether the address is urban or rural. For example, if the user enters `T2N 1N4` then your program should indicate that the postal code is for an urban address in Alberta. If the user enters `X0A 1B2` then your program should indicate that the postal code is for a rural address in Nunavut or Northwest Territories. Use a dictionary to map from the first character of the postal code to the province or territory name. Display a meaningful error message if the postal code begins with an invalid character, or if the second character in the postal code is not a digit.

Exercise 141: Write out Numbers in English

(*65 Lines*)

While the popularity of cheques as a payment method has diminished in recent years, some companies still issue them to pay employees or vendors. The amount being paid normally appears on a cheque twice, with one occurrence written using digits, and the other occurrence written using English words. Repeating the amount in two different forms makes it much more difficult for an unscrupulous employee or vendor to modify the amount on the cheque before depositing it.

In this exercise, your task is to create a function that takes an integer between 0 and 999 as its only parameter, and returns a string containing the English words for that number. For example, if the parameter to the function is 142 then your function should return "`one hundred forty two`". Use one or more dictionaries to implement your solution rather than large if/elif/else constructs. Include a main program that reads an integer from the user and displays its value in English words.

Exercise 142: Unique Characters

(*Solved, 16 Lines*)

Create a program that determines and displays the number of unique characters in a string entered by the user. For example, `Hello, World!` has 10 unique characters while `zzz` has only one unique character. Use a dictionary or set to solve this problem.

Exercise 143: Anagrams

(*Solved, 39 Lines*)

Two words are anagrams if they contain all of the same letters, but in a different order. For example, "evil" and "live" are anagrams because each contains one "e", one "i", one "l", and one "v". Create a program that reads two strings from the user, determines whether or not they are anagrams, and reports the result.

Exercise 144: Anagrams Again

(*48 Lines*)

The notion of anagrams can be extended to multiple words. For example, "William Shakespeare" and "I am a weakish speller" are anagrams when capitalization and spacing are ignored.

Extend your program from Exercise 143 so that it is able to check if two phrases are anagrams. Your program should ignore capitalization, punctuation marks and spacing when making the determination.

Exercise 145: Scrabble™ Score

(*Solved, 18 Lines*)

In the game of Scrabble™, each letter has points associated with it. The total score of a word is the sum of the scores of its letters. More common letters are worth fewer points while less common letters are worth more points. The points associated with each letter are shown below:

Points	Letters
1	A, E, I, L, N, O, R, S, T and U
2	D and G
3	B, C, M and P
4	F, H, V, W and Y
5	K
8	J and X
10	Q and Z

Write a program that computes and displays the Scrabble™ score for a word. Create a dictionary that maps from letters to point values. Then use the dictionary to compute the score.

A Scrabble™ board includes some squares that multiply the value of a letter or the value of an entire word. We will ignore these squares in this exercise.

Exercise 146: Create a Bingo Card

(*Solved, 58 Lines*)

A Bingo card consists of 5 columns of 5 numbers which are labelled with the letters `B`, `I`, `N`, `G` and `O`. There are 15 numbers that can appear under each letter. In particular, the numbers that can appear under the `B` range from 1 to 15, the numbers that can appear under the `I` range from 16 to 30, the numbers that can appear under the `N` range from 31 to 45, and so on.

Write a function that creates a random Bingo card and stores it in a dictionary. The keys will be the letters `B`, `I`, `N`, `G` and `O`. The values will be the lists of five numbers that appear under each letter. Write a second function that displays the Bingo card with the columns labelled appropriately. Use these functions to write a program that displays a random Bingo card. Ensure that the main program only runs when the file containing your solution has not been imported into another program.

You may be aware that Bingo cards often have a "free" space in the middle of the card. We won't consider the free space in this exercise.

Exercise 147: Checking for a Winning Card

(*102 Lines*)

A winning Bingo card contains a line of 5 numbers that have all been called. Players normally record the numbers that have been called by crossing them out or marking them with a Bingo dauber. In this exercise we will mark that a number has been called by replacing it with a 0 in the Bingo card dictionary.

Write a function that takes a dictionary representing a Bingo card as its only parameter. If the card contains a line of five zeros (vertical, horizontal or diagonal) then your function should return `True`, indicating that the card has won. Otherwise the function should return `False`.

Create a main program that demonstrates your function by creating several Bingo cards, displaying them, and indicating whether or not they contain a winning line. You should demonstrate your function with at least one card with a horizontal line, at least one card with a vertical line, at least one card with a diagonal line, and at least one card that has some numbers crossed out but does not contain a winning line. You will probably want to import your solution to Exercise 146 when completing this exercise.

Hint: Because there are no negative numbers on a Bingo card, finding a line of 5 zeros is equivalent to finding a line of 5 entries that sum to zero. You may find the summation problem easier to solve.

Exercise 148: Play Bingo

(*88 Lines*)

In this exercise you will write a program that simulates a game of Bingo for a single card. Begin by generating a list of all of the valid Bingo calls (B1 through O75). Once the list has been created you can randomize the order of its elements by calling the `shuffle` function in the `random` module. Then your program should consume calls out of the list and cross out numbers on the card until the card contains a winning line. Simulate 1,000 games and report the minimum, maximum and average number of calls that must be made before the card wins. You may find it helpful to import your solutions to Exercises 146 and 147 when completing this exercise.