# Dictionary Solutions

# 14

## Solution to Exercise 128: Reverse Lookup

```
##
# Conduct a reverse lookup on a dictionary, finding all of the keys that map to the provided value.
#

## Conduct a reverse lookup on a dictionary
#  @param data the dictionary to perform the reverse lookup on
#  @param value the value to search for in the dictionary
#  @return a list (possibly empty) of keys from data that map to value
def reverseLookup(data, value):
  # Construct a list of the keys that map to value
  keys = []

  # Check each key, adding it to keys if the values match
  for key in data:
    if data[key] == value:
      keys.append(key)

  # Return the list of keys
  return keys

# Demonstrate the reverseLookup function
def main():
  # A dictionary mapping 4 French words to their English equivalents
  frEn = {"le" : "the", "la" : "the", "livre" : "book", "pomme" : "apple"}

  # Demonstrate the reverseLookup function with 3 cases: One that returns
  # multiple keys, one that returns one key, and one that returns no keys
  print("The french words for 'the' are: ", reverseLookup(frEn, "the"))
  print("Expected: ['le', 'la']")
  print()
  print("The french word for 'apple' is: ", reverseLookup(frEn, "apple"))
  print("Expected: ['pomme']")
  print()
```

Each key in a dictionary must be unique. However, values may be repeated. As a result, performing a reverse lookup may identify zero, one or several keys that match the provided value.



B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_14

```
  print("The french word for 'asdf' is: ", reverseLookup(frEn, "asdf"))
  print("Expected: []")

# Only call the main function if this file has not been imported
if __name__ == "__main__":
  main()
```

## Solution to Exercise 129: Two Dice Simulation

```
##
# Simulate rolling two dice many times and compare the simulated results
# to the results expected by probability theory.
#
from random import randrange

NUM_RUNS = 1000
D_MAX = 6

## Simulate rolling two six-sided dice
# @return the total of rolling two simulated dice
def twoDice():
  # Simulate two dice
  d1 = randrange(1, D_MAX + 1)
  d2 = randrange(1, D_MAX + 1)

  # Return the total
  return d1 + d2

# Simulate many rolls and display the result
def main():
  # Create a dictionary of expected proportions
  expected = {2: 1/36, 3: 2/36, 4: 3/36, 5: 4/36, \
              6: 5/36, 7: 6/36, 8: 5/36, 9: 4/36, \
              10: 3/36, 11: 2/36, 12: 1/36}

  # Create a dictionary that maps from the total of
  # two dice to the number of occurrences
  counts = {2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, \
            8: 0, 9: 0, 10: 0, 11: 0, 12: 0}

  # Simulate NUM_RUNS rolls, and count each roll
  for i in range(NUM_RUNS):
    t = twoDice()
    counts[t] = counts[t] + 1

  # Display the simulated proportions and the expected proportions
  print("Total   Simulated   Expected")
  print("          Percent    Percent")
  for i in sorted(counts.keys()):
    print("%5d %11.2f   %8.2f" % \
          (i, counts[i] / NUM_RUNS * 100, expected[i] * 100))
```

Each dictionary is initialized so that it has keys 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 and 12. In the `expected` dictionary, the value is initialized to the probability that each key will occur when two 6-sided dice are rolled. In the `counts` dictionary, each value is initialized to 0. The values in `counts` are increased as the simulation runs.

```
# Call the main function
main()
```

## Solution to Exercise 134: Unique Characters

```
##
# Compute the number of unique characters in a string using a dictionary.
#

# Read the string from the user
s = input("Enter a string: ")

# Add each character to a dictionary with a value of True. Once we are done the number
# of keys in the dictionary will be the number of unique characters in the string.
characters = {}
for ch in s:
  characters[ch] = True
```

> Every key in a dictionary must have a value associated with it. However, in this solution the value is never used. As a result, we have elected to associate `True` with each key, but any other value could have been used instead of `True`.

```
# Display the result
print("That string contained", len(characters), "unique character(s).")
```

> The `len` function returns the number of keys in a dictionary.

## Solution to Exercise 135: Anagrams

```
##
# Determine whether or not two strings are anagrams and report the result.
#

## Compute the frequency distribution for the characters in a string
# @param s the string to process
# @return a dictionary mapping each character to its count
def characterCounts(s):
  # Create a new, empty dictionary
  counts = {}

  # Update the count for each character in the string
  for ch in s:
    if ch in counts:
      counts[ch] = counts[ch] + 1
    else:
      counts[ch] = 1
```

```
  # Return the result
  return counts

# Determine if two strings entered by the user are anagrams
def main():
  # Read the strings from the user
  s1 = input("Enter the first string: ")
  s2 = input("Enter the second string: ")

  # Get the character counts for each string
  counts1 = characterCounts(s1)
  counts2 = characterCounts(s2)

  # Display the result
  if counts1 == counts2:
    print("Those strings are anagrams.")
  else:
    print("Those strings are not anagrams.")

# Call the main function
main()
```

Two dictionaries are equal if and only if both dictionaries have the same keys and for every key, $k$, the value associated with $k$ is the same in both dictionaries.

## Solution to Exercise 137: Scrabble™ Score

```
##
# Use a dictionary to compute the Scrabble™ score for a word.
#

# Initialize the dictionary so that it maps from letters to points
points = {"A": 1, "B": 3, "C": 3, "D": 2, "E": 1, "F": 4, \
          "G": 2, "H": 4, "I": 1, "J": 2, "K": 5, "L": 1, \
          "M": 3, "N": 1, "O": 1, "P": 3, "Q": 10, "R": 1, \
          "S": 1, "T": 1, "U": 1, "V": 4, "W": 4, "X": 8,
          "Y": 4, "Z": 10}

# Read a word from the user
word = input("Enter a word: ")

# Compute the score for the word
uppercase = word.upper()
score = 0
for ch in uppercase:
  score = score + points[ch]

# Display the result
print(word, "is worth", score, "points.")
```

The word is converted to uppercase so that the user will get a correct response when they enter the word in upper, mixed or lowercase. This could also be accomplished by adding all of the lowercase letters to the dictionary.

## Solution to Exercise 138: Create a Bingo Card

```
##
#  Create and display a random Bingo card.
#
from random import randrange

NUMS_PER_LETTER = 15

# Create a bingo card with randomly generated numbers
# @return a dictionary representing the card where the keys are the strings
#       ''B'', ''I'', ''N'', ''G'', and ''O'', and the values are lists of the
#       numbers that appear under each letter from top to bottom
def createCard():
  card = {}

  # The range of integers that can be generated for the current letter
  lower = 1
  upper = 1 + NUMS_PER_LETTER

  # For each of the five letters
  for letter in ["B", "I", "N", "G", "O"]:
    # Start with an empty list for the letter
    card[letter] = []

    # Keep generating random numbers until we have 5 unique ones
    while len(card[letter]) != 5:
      next_num = randrange(lower, upper)
      # Ensure that we do not include any duplicate numbers
      if next_num not in card[letter]:
        card[letter].append(next_num)

    # Update the range of values that will be generated for the next letter
    lower = lower + NUMS_PER_LETTER
    upper = upper + NUMS_PER_LETTER

  # Return the generated card
  return card

# Display a bingo card with nice formatting
# @param card the bingo card to display
# @return (None)
def displayCard(card):
  # Display the headings
  print("B  I  N  G  O")

  # Display the numbers on the card
  for i in range(5):
    for letter in ["B", "I", "N", "G", "O"]:
      print("%2d " % card[letter][i], end="")
    print()
```

```
# Generate a random Bingo card and display it
def main():
  card = createCard()
  displayCard(card)

# Call the main program only if this file hasn't been imported
if __name__ == "__main__":
  main()
```