Recursion Solutions

16

Solution to Exercise 164: Total the Values

```
##
#  Compute the total of a collection of numbers entered by the user.  The user
#  will enter a blank line to indicate that no further numbers will be entered.
#

# Compute the total of all the numbers entered by the user until the user enters a blank line.
# @return the total of the entered values
def readAndTotal():
  # Read a value from the user
  line = input("Enter a number (blank to quit): ")

  # Base case: The user entered a blank line so the total is 0
  if line == "":
    return 0
  else:
    # Recursive case: Convert the current line to a number and use recursion to read the next line
    return float(line) + readAndTotal()

# Read a collection of numbers from the user and display the total
def main():
  # Read the values from the user and compute the total
  total = readAndTotal()

  # Display the result
  print("The total of all those values is", total)

# Call the main function
main()
```

Solution to Exercise 167: Recursive Palindrome

```
##
#  Determine whether or not a string entered by the user is a palindrome using recursion.
#
```

© Springer International Publishing Switzerland 2014

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_16

```
## Determine whether or not a string is a palindrome
#  @param s the string to check
#  @return True if the string is a palindrome, False otherwise
def isPalindrome(s):
  # Base case: the empty string is a palindrome.  So is a string containing only 1 character.
  if len(s) <= 1:
    return True

  # Recrusive case: The string is a palindrome only if the first and last
  # characters match, and the rest of the string is a palindrome
  return s[0] == s[len(s) - 1] and isPalindrome(s[1 : len(s) - 1])

# Check whether or not a string entered by the user is a palindrome
def main():
  # Read the input from the user
  line = input("Enter a string: ")

  # Check the status and display the result
  if isPalindrome(line):
    print("That was a palindrome!")
  else:
    print("That is not a palindrome.")

# Call the main function
main()
```

Solution to Exercise 169: String Edit Distance

```
##
# Compute and display the edit distance between two strings.
#

## Compute the edit distance between two strings as a count of the minimum number of insert,
#  delete and substitute operations needed to transform one string into the other.
#  @param s the first string
#  @param t the second string
#  @param the edit distance between the strings
def editDistance(s, t):
  # If one string is empty, then the edit distance is one insert operation
  # for each letter in the other string
  if len(s) == 0:
    return len(t)
  elif len(t) == 0:
    return len(s)
  else:
    cost = 0
    # If the last characters are not equal, set cost to 1
    if s[len(s) - 1] != t[len(t) - 1]:
      cost = 1
```

```
    # Compute three distances
    d1 = editDistance(s[0 : len(s) - 1], t) + 1
    d2 = editDistance(s, t[0 : len(t) - 1]) + 1
    d3 = editDistance(s[0 : len(s) - 1] , t[0 : len(t) - 1]) + cost

    # Return the minimum distance
    return min(d1, d2, d3)

# Compute the edit distance between two strings entered by the user
def main():
  # Read input from the user
  s1 = input("Enter a string: ")
  s2 = input("Enter another string: ")

  # Compute and display the result
  print("The edit distance between %s and %s is %d." % \
        (s1, s2, editDistance(s1, s2)))

# Call the main function
main()
```

Solution to Exercise 172: Element Sequences

```
##
# Determine the longest sequence of elements that can follow an element
# entered by the user where each element in the sequence begins with
# the same letter as the last letter of its predecessor.
#
from sys import *

ELEMENTS_FNAME = "../Data/Elements.csv"

# Find the longest sequence of words, beginning with start, where each word
# begins with the last letter of its predecessor.
# @param start the first word in the sequence
# @param the list of words that can occur in the sequence
# @return the longest list of words beginning with start that meets the
#       letter constraints outlined previously
def longestSequence(start, words):
  # Base case: If start is empty then the list of words is empty
  if start == "":
    return []

  # Find the best (longest) list of words by checking each possible word
  # that could appear next in the sequence
  best = []
  last_letter = start[len(start) - 1].lower()
  for i in range(0, len(words)):
    first_letter = words[i][0].lower()
```

```
      # If the first letter of the next word matches the last letter of the previous word
      if first_letter == last_letter:
        # Use recursion to find a candidate sequence of words
        candidate = longestSequence(words[i], \
                    words[0 : i] + words[i + 1 : len(words)])
        # Save the candidate if it is better than the best sequence that we have seen previously
        if len(candidate) > len(best):
          best = candidate

    # Return the best candidate sequence, preceeded by the starting word
    return [start] + best

# Load the names of all of the elements from the elements file
# @return a list of all the element names
def loadNames():
    names = []

    # Open the element data set
    inf = open(ELEMENTS_FNAME, "r")

    # Load each line, storing the element name in a list
    for line in inf:
        line = line.rstrip()
        parts = line.split(",")
        names.append(parts[2])

    # Close the file and return the list
    inf.close()
    return names

# Display a longest sequence of elements starting with an element entered by the user
def main():
    # Load all of the element names
    names = loadNames()

    # Read the starting element and capitalize it
    start = input("Enter the name of an element: ")
    start = start.capitalize()

    # Remove the starting element from the list of elements
    names.remove(start)

    # Find a longest sequence starting with start
    sequence = longestSequence(start, names)

    # Display the sequence of elements
    print("A longest sequence that starts with", start, "is:")
    for element in sequence:
        print(" ", element)

# Call the main function
main()
```

Solution to Exercise 174: Run-Length Encoding

```
##
# Perform run-length encoding on a string of values using recursion.
#

## Perform run-length encoding on a string or list of values
# @param data the string or list to encode
# @return a list where the elements at even positions are data values and the
#       elements at odd positions are counts of the number of times that
#       the data value before it should be replicated.
def encode(data):
  # If data is empty then no encoding work is necessary
  if len(data) == 0:
    return []
```

If we performed the comparison `data == ""` then this function would only work for strings. If we performed the comparison `data == []` then it would only work for lists. Checking the length of the parameter allows the function to work for both strings and lists.

```
  # Find the index of the first item that is not the same as the item at position 0 in data
  index = 1
  while index < len(data) and data[index] == data[index - 1]:
    index = index + 1

  # Encode the current character group
  current = [data[0], index]

  # Use recursion to process the characters from index to the end of the string
  return current + encode(data[index : len(data)])

# Demonstrate the encode function
def main():
  # Read a string from the user
  s = input("Enter some characters: ")

  # Display the result
  print("When those characters are run-length encoded, the result is:")
  print(encode(s))

# Call the main function
main()
```