HOMEWORK WEEK 12

Introduction to Programming with Python (926)

Elena Sizikova

Wednesday
Mar 11, 2015 - May 27, 2015
7:30-9:00 PM ET, 4:30-6:00 PM PT

Homework: Week 12

Your writing problem response has been graded.
Past Due **Jun 4**.

Readings

Lesson 12: Transcript

Challenge Problems

Problem 1 – Graded – Technical: 7 / 7 – Style: 1 / 1 (8071)

Problem: Report an Error

For the final two weeks of the course, you will be working on a final project. You should choose **one** of the following three games to implement in Python. (You can do more than one if you want, but you should only submit one as your project.) Pick the one that you find most interesting or most challenging.

For week 12, you should submit the project. You should also include sample runs of your code, a description (in English) of your algorithm, and what you did to test that your program works properly. Also if you made any modifications to the game, you should describe them as well.

For each of these games, you shouldn't feel constrained by the descriptions given below. Feel free to add or delete specific aspects of the game as you see fit. (But in both your week 11 and week 12 submissions, you should describe what it is that you've altered.) Also you don't have to exactly match the sample runs of the game that are shown -- feel free to be creative and to add features not shown.

You are allowed to work with your classmates if you like. You can use the message board and/or get together in the classroom during the week to collaborate. However, if you work with others, you should credit them appropriately when turning in your project. (In other words, say with whom you worked and what part(s) of the project each person did.) Also, please feel free to ask for hints or suggestions on the message board.

The projects are listed roughly in order of difficulty. (To be fair, this can be a bit difficult to judge.)

Project #1: Mastermind

Mastermind is a game for 2 players. The commercial version of the game uses six colors, which vary from game to game; for our purposes we'll use red, yellow, blue, green, orange, and purple. One player (the **codemaker**) thinks of a secret code, which is any 4 colors in some order (in the easy version of the game, the 4 colors all have to be different; in the harder version, a color is allowed to repeat one or more times). The other player (the **codebreaker**) has 10 attempts to guess the code.

For each guess, the codemaker tells the codebreaker how many colors are correct and in the correct location, and how many colors are correct but in the wrong location. The guesser is **not** told exactly which colors are correct. In the commercial version of the game, black and white pegs are used for this: a black peg means a color correct and in the correct location, a white peg means a color correct but not in the correct location. You can use black and white as a shorthand for your responses.

Here's an example: suppose the code is red-blue-red-yellow.

If a guess is green-blue-purple-purple, the response would be **1 black 0 white** (meaning that one peg---the blue one---is correct and in the correct location, and the rest are incorrect).

If a guess is red-yellow-green-purple, the response would be **1 black 1 white** (the red is correct and in the correct location, the yellow is correct but in the wrong location).

If a guess is red-red-blue-orange, the response would be **1 black 2 white**.

If a guess is blue-green-blue-red, the response would be **0 black 2 white** (only one of the two blue guessed gets a white peg, since there is only one blue in the actual code).

Implement this game where the computer randomly generates a code, and the user plays as the codebreaker. The codebreaker wins if he or she gets the code in 10 guesses or less. You can decide whether to implement the easy or hard version of the game (or both).

A sample run:

```
I've got a code of length 4
It uses colors in rybgop
Guess #1 -- enter your guess: rybg
1 black 1 white
Guess #2 -- enter your guess: ryop
2 black 1 white
Guess #3 -- enter your guess: rpyo
1 black 2 white
Guess #4 -- enter your guess: ryog
1 black 2 white
Guess #5 -- enter your guess: ropy
2 black 1 white
Guess #6 -- enter your guess: ropg
2 black 2 white
Guess #7 -- enter your guess: rogp
You cracked the code!
```

Project #2: Connect Four

Connect Four is a tic-tac-toe-based game for 2 players. In the commercial version of the game, players take turns dropping colored checkers (one player is red, the other is black) into one of 7 columns of the game device. The checkers will fall to the bottom of the column. Each column can hold up to 6 checkers. The first player to get 4 checkers of his or her color in a row (in any direction, including diagonally) is the winner. If both players play all 21 of their checkers with neither player getting 4-in-a-row, the game is a draw.

Implement Connect Four so that two human players can play each other. We suggest that you use **X** and **O** for the two players.

A sample run:

```
Player X, enter your name: Dave
Player O, enter your name: Niki

0 1 2 3 4 5 6
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .

Dave, you're X. What column do you want to play in? 2

0 1 2 3 4 5 6
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . X . . . .

Niki, you're O. What column do you want to play in? 3

0 1 2 3 4 5 6
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . X O . . .

Dave, you're X. What column do you want to play in? 3

0 1 2 3 4 5 6
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . X . . .
. . X O . . .

Niki, you're O. What column do you want to play in? 4

0 1 2 3 4 5 6
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . X . . .
. . X O O . .
```

```
Dave, you're X. What column do you want to play in? 1

0 1 2 3 4 5 6
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . X . . .
. X X O O . .

Niki, you're O. What column do you want to play in? 4

0 1 2 3 4 5 6
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . X O . .
. X X O O . .

Dave, you're X. What column do you want to play in? 2

0 1 2 3 4 5 6
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . X X O . .
. X X O O . .

Niki, you're O. What column do you want to play in? 4

0 1 2 3 4 5 6
. . . . . . .
. . . . . . .
. . . . . . .
. . . . O . .
. . X X O . .
. X X O O . .

Dave, you're X. What column do you want to play in? 4

0 1 2 3 4 5 6
. . . . . . .
. . . . . . .
. . . . X . .
. . . . O . .
. . X X O . .
. X X O O . .

Niki, you're O. What column do you want to play in? 5

0 1 2 3 4 5 6
. . . . . . .
```

```
. . . . . .
. . . . X . .
. . . . O . .
. . X X O . .
. X X O O O .

Dave, you're X. What column do you want to play in? 6

0 1 2 3 4 5 6
. . . . . .
. . . . . .
. . . . X . .
. . . . O . .
. . X X O . .
. X X O O O X

Niki, you're O. What column do you want to play in? 3

0 1 2 3 4 5 6
. . . . . .
. . . . . .
. . . . X . .
. . . O O . .
. . X X O . .
. X X O O O X

Dave, you're X. What column do you want to play in? 5

0 1 2 3 4 5 6
. . . . . .
. . . . . .
. . . . X . .
. . . O O . .
. . X X O X .
. X X O O O X

Niki, you're O. What column do you want to play in? 5

0 1 2 3 4 5 6
. . . . . .
. . . . . .
. . . . X . .
. . . O O O .
. . X X O X .
. X X O O O X

Dave, you're X. What column do you want to play in? 2

0 1 2 3 4 5 6
. . . . . .
. . . . . .
. . . . X . .
. . X O O O .
. . X X O X .
```

```
. X X O O O X

Niki, you're O. What column do you want to play in? 2

0 1 2 3 4 5 6
. . . . . . .
. . . . . . .
. . O . X . .
. . X O O O .
. . X X O X .
. X X O O O X

Congratulations, Niki, you won!
```

Project #3: Boggle

Boggle is a game played with 16 6-sided dice with letters on them. The sixteen dice have sides:

```
AAEEGN ELRTTY AOOTTW ABBJOO
EHRTVW CIMOTU DISTTY EIOSST
DELRVY ACHOPS HIMNQU EEINSU
EEGHNW AFFKPS HLNNRZ DELIRX
```

(The actual commercial game has a "QU" on one side instead of just "Q", but we've changed the game slightly to make it easier to implement.) The dice are randomly rolled and arranged in a 4×4 grid. Shown below is an example:

```
T  L  H  O
```

```
V  M  T  O
Y  S  S  A
O  K  E  D
```

The goal is to form words (with 3 or more letters) by starting at any letter and moving one letter at a time in any direction (including diagonals). You can't use the same letter in the grid twice in the same word. Some valid words in the above grid are HOT, TOO, HOOT, HOOTS, SOOT, SKY, DESK, DESKS, SAT, ASK, ASKED, SOY, YOKE, YOKES, YOKED, TOAD, TOADS, and there are probably more. But TOOTH and TOAST are not valid since each uses the T twice, and neither is SEED because you can't "stay" on a letter. If the same word appears in more than one place in the grid (such as HOT in the above example), it only counts once.

In the commercial game, each player gets 3 minutes to find as many words as they can. Words score according to the chart below:

```
Letters in word  3  4  5  6  7  8 or more
          Score  1  1  2  3  5  11
```

Use the `wordlist.txt` file at http://aops-docs.s3.amazonaws.com/python1/wordlist.txt to check the validity of words.

Write a solitare version of Boggle. Your game should generate a grid and then let the player find as many words as he or she can. You should check that each word is in the dictionary and is in the grid. After the player gives up, the program should print the player's score. Also, optionally, have the computer print the longest possible word in the grid.

A sample run:

```
O    E    B    I

T    A    N    T

L    O    R    T

G    X    N    A

Enter your word (leave blank to quit): bane
BANE is a valid word!

O    E    B    I

T    A    N    T

L    O    R    T

G    X    N    A

Enter your word (leave blank to quit): tart
TART is a valid word!

O    E    B    I

T    A    N    T

L    O    R    T
```

```
G    X    N    A

Enter your word (leave blank to quit): rox
ROX is NOT in the dictionary!

O    E    B    I

T    A    N    T

L    O    R    T

G    X    N    A

Enter your word (leave blank to quit): tag
TAG is NOT in the grid!

O    E    B    I

T    A    N    T

L    O    R    T

G    X    N    A

Enter your word (leave blank to quit): baron
BARON is a valid word!

O    E    B    I

T    A    N    T

L    O    R    T

G    X    N    A

Enter your word (leave blank to quit): tartan
TARTAN is a valid word!

O    E    B    I

T    A    N    T

L    O    R    T

G    X    N    A

Enter your word (leave blank to quit): lox
LOX is a valid word!

O    E    B    I

T    A    N    T
```

```
L    O    R    T

G    X    N    A

Enter your word (leave blank to quit):
Here's your score:
BANE scores 1
TART scores 1
BARON scores 2
TARTAN scores 3
LOX scores 1
TOTAL SCORE: 8
Thanks for playing!
Let me see...I bet I can find a long word.
Please give me a moment to think...
I found BARONET
```

Solution:
Note: for all of these projects, your answer might be quite different from ours -- indeed, I hope that you implemented features that I might not have thought of!

These will all run better if you copy them as an IDLE module and run them in IDLE. (All of the input and output doesn't work quite so well on the website.) Boggle won't work at all unless you have the wordlist.txt file in the same window as your .py file.

Project #1: Mastermind

```
import random

def return_matches(guess,code):
    '''return_matches(guess,code) -> (int,int)
    returns mathces of guess to code
    first int is "black" (exact matches)
    second int is "white" (right color, wrong position)'''
    black,white = 0,0  # initialize match variables
    # use these list to keep track of matches
    guessMatched = [False]*len(guess)
    codeMatched = [False]*len(guess)
    # determine exact matches
    for n in range(len(guess)):
        if guess[n] == code[n]:
            guessMatched[n] = True
            codeMatched[n] = True
            black += 1
    # determine position matches
    for n in range(len(guess)):
        if not guessMatched[n]: # not a black match, check for whit
            for char in range(len(guess)):  # loop through the code
                if not codeMatched[char] and guess[n]==code[char]:
                    # found a match -- update and break
                    codeMatched[char] = True
                    white += 1
                    break
    return (black,white)

def play_mastermind(codeLength,numGuesses,difficulty):
```

```
    '''play_mastermind(codeLength,numGuesses) -> None
    play a game of mastermind
    codeLength is the length of the code
    numGuesses is the number of guesses allowed
    difficulty is 'easy' or 'hard' '''
    colors = 'rybgop'
    code = ''
    winner = False
    # create a code
    for i in range(codeLength):
        color = colors[random.randrange(len(colors))]
        while difficulty == 'easy' and color in code:
            # duplicate color, pick another
            color = colors[random.randrange(len(colors))]
        code += color
    # play the game
    print("I've got a code of length "+str(codeLength))
    print("It uses colors in "+colors)
    for turn in range(1,numGuesses+1):
        legalGuess = False
        while not legalGuess:
            guess = input("Guess #"+str(turn)+" -- enter your guess
            legalGuess = True
            # check for legal guess
            if len(guess) == codeLength:
                # check for legal colors
                for index in range(codeLength):
                    if guess[index] not in colors:
                        legalGuess = False
            else:
                legalGuess = False
            if not legalGuess:
                print("That's not a legal guess!")
        (black,white) = return_matches(guess,code)
        if black==4:
            print("You cracked the code!")
            winner = True
            break
        print(str(black)+" black "+str(white)+ " white")
    if not winner:
        print("Sorry, you ran out of guesses!")
        print("The code was: "+str(code))

play_mastermind(4,10,'hard')
```

Run Pop Out ↗ Reset

Project #2: Connect Four

```
def print_board(board):
    '''print_board(board) -> str
    returns a string that represents the Connect Four
    board given by the input list'''
    boardString = '\n'
    # add the column numbers to the output string
    for col in range(7):
        boardString += str(col) + ' '
    boardString += '\n'
```

```
    # loop through the input list
    # add the appropriate characters to the output string
    for row in range(6):
        for col in range(7):
            boardString += board[row][col] + ' '
        boardString += '\n'
    return boardString

def check_line_for_win(board,r,c,dr,dc,piece):
    '''check_line_for_win(board,r,c,dr,dc,piece) -> bool
    returns True if piece has a winning line
    starting at row r, col c and moving in the direction
      given by dr and dc'''
    for i in range(4):
        if board[r+i*dr][c+i*dc] != piece:
            return False
    return True

def check_for_win(board,piece):
    '''check_for_win(board,piece) -> bool
    returns True if piece has a winning line on the board'''
    # check horizontal
    for row in range(6):
        for col in range(4):
            if check_line_for_win(board,row,col,0,1,piece):
                return True
    # check vertical
    for row in range(3):
        for col in range(7):
            if check_line_for_win(board,row,col,1,0,piece):
                return True
    # check NW-SE diag
    for row in range(3):
        for col in range(4):
            if check_line_for_win(board,row,col,1,1,piece):
                return True
    # check NE-SW diag
    for row in range(3,6):
        for col in range(4):
            if check_line_for_win(board,row,col,-1,1,piece):
                return True
    return False


def play_connect_four():
    # initialize
    playerNames = []
    playerNames.append(input("Player X, enter your name: "))
    playerNames.append(input("Player O, enter your name: "))
    pieces = ['X','O']
    turn = 0  # whose turn is it
    turns = 0 # number of moves in the game
    # set up a blank board
    boardRow = ['.']*7
    board = []
    for row in range(6):
        board.append(boardRow[:])
    # initialize the column heights
    #  columnHeights[n] is the number of pieces
    #  played in column n
columnHeights = [0]*7
```

```
 71
    # play the game
    while turns < 42:
        print(print_board(board))
        legalPlay = False
        while not legalPlay:
            play = input(playerNames[turn]+", you're "+pieces[turn
            if not play.isdigit():
                print("That's not a valid column -- please choose
            else:
                play = int(play)
                if play < 0 or play > 6:
                    print("That's not a valid column -- please cho
                elif columnHeights[play] == 6:
                    print("That column is full -- please choose an
                else:
                    legalPlay = True
        # make the play
        height = 5-columnHeights[play]
        board[height][play] = pieces[turn]
        columnHeights[play] += 1
        # check for winner
        winner = check_for_win(board,pieces[turn])
        if winner:
            print(print_board(board))
            print("Congratulations, "+str(playerNames[turn])+", yo
            break
        turn = (turn + 1) % 2
        turns += 1
    if turns == 42:  # tie game
        print(print_board(board))
        print("It's a tie!")

play_connect_four()
```

Run Pop Out ↗ Reset

Project #3: Boggle

We have two versions. The first version uses text only, and prints the grid after every user entry.

```
import random

def roll_boggle_die(die):
    '''roll_boggle_die(die) -> str
    return a character'''
    letter = die[random.randrange(len(die))]
    return letter

def create_boggle_grid(boggledice):
    '''create_boggle_grid(boggledice) -> list
    returns a list that's a boggle grid'''
    diceList = boggledice[:]
    gridList = []
    while len(diceList) > 0:
        # pick a random die
        die = diceList[random.randrange(len(diceList))]
        # roll it and add it to the grid
```

```
        gridList.append(roll_boggle_die(die))
        # remove it from the list so we don't use it again
        diceList.remove(die)
    return gridList

def print_boggle_grid(gridList):
    '''print_boggle_grid(gridList) -> str
    returns string that represents the grid'''
    gridString = '\n'
    for index in range(len(gridList)):
        gridString += gridList[index]
        if index%4 == 3:  # go to the next row
            gridString += '\n\n'
        else:  # go to the next column
            gridString += '\t'
    return gridString[:-1]

def convert_to_grid(num):
    '''convert_to_grid(num) -> (int,int)
    converts num 0-15 to grid coordinates'''
    return(num//4, num%4)

def adjacent_in_grid(pos1,pos2):
    '''adjacent_in_grid(pos1,pos2) -> bool
    returns True if they're adjacent'''
    # convert to coordinates
    coord1 = convert_to_grid(pos1)
    coord2 = convert_to_grid(pos2)
    if coord1==coord2:  # same square
        return False
    # both rows and columns must be at most 1 apart
    return abs(coord1[0]-coord2[0])<=1 and abs(coord1[1]-coord2[1]

def letter_list(letter,gridList):
    '''letter_list(letter,gridList) -> list
    returns list of indices of letter in gridList'''
    letterList = []
    for index in range(len(gridList)):
        if gridList[index] == letter:
            letterList.append(index)
    return letterList

def create_paths(listOfLists):
    '''create_paths(listOfLists) -> list
    returns a list of the paths of the lists in the argument
    omits paths that go to the same point twice'''
    paths = [[]]  # start with an empty path
    for nextList in listOfLists:
        newpaths = []
        for priorPath in paths:
            # extend each prior path by each number in the next li
            for num in nextList:
                if num not in priorPath: # only extend if not a du
                    newpaths.append(priorPath + [num])
        paths = newpaths  # reset the current paths
    return paths

def valid_word(word,gridList):
    '''valid_word(word,gridList) -> bool
    returns True if word is a valid word in gridList'''
letterLists = []
```

```
78  79      # make a list of where all the letters in word appear in t
80      for char in word:
81          letterLists.append(letter_list(char,gridList))
82          if letterLists[-1] == []:  # letter doesn't exist
83              return False
84      # make all the possible paths of the letters
85      paths = create_paths(letterLists)
86      for path in paths:  # check if the path is a legal path
87          pathOK = True
88          for index in range(len(path)-1):
89              # make sure each letter is adjacent to the next letter
90              if not adjacent_in_grid(path[index],path[index+1]):
91                  pathOK = False
92                  break
93          if pathOK:
94              return True
95      return False
96
97  def play_boggle():
98      '''play_boggle() -> None
99      plays a round of Boggle'''
100     boggledice = [
101         'AAEEGN','ELRTTY','AOOTTW','ABBJOO',
102         'EHRTVW','CIMOTY','DISTTY','EIOSST',
103         'DELRVY','ACHOPS','HIMNQU','EEINSU',
104         'EEGHNW','AFFKPS','HLNNRZ','DELIRX'
105         ]
106     grid = create_boggle_grid(boggledice)
107     wordList = []
108     # create dictionary of legal words
109     wordFile = open('wordlist.txt','r')
110     legalWords = wordFile.readlines()
111     wordFile.close
112     # get words from user
113     while True:
114         print(print_boggle_grid(grid))
115         word = input("Enter your word (leave blank to quit): ").up
116         if word == '':
117             break  # done guessing
118         if word.lower()+'\n' not in legalWords:  # not a valid wor
119             print(word+' is NOT in the dictionary!')
120         elif valid_word(word,grid):
121             if len(word) < 3:  # too short
122                 print(word+' is not long enough.')
123             elif word in wordList:  # already found
124                 print('You already found '+word)
125             else:  # valid -- add to list
126                 print(word+' is a valid word!')
127                 wordList.append(word)
128         else:  # not in the grid
129             print(word+' is NOT in the grid!')
130     # compute score
131     print("Here's your score:")
132     score = 0
133     scoreChart = (0,0,0,1,1,2,3,5)
134     for word in wordList:
135         length = len(word)
136         if length >= 8:
137             wordScore = 11
138         else:
            wordScore = scoreChart[length]
```

```
            print(word+" scores "+str(wordScore))
        score += wordScore
    print("TOTAL SCORE: "+str(score))
    print("Thanks for playing!")
    # have computer look for longest word
    print("Let me see...I bet I can find a long word.")
    print("Please give me a moment to think...")
    maxLength = 2
    for word in legalWords:
        word = word.strip('\n').upper()
        if len(word) > maxLength and valid_word(word,grid):
            longestWord = word
            maxLength = len(word)
    print("I found "+longestWord)

play_boggle()
```

Run Pop Out ↗ Reset

The second version uses a turtle to draw the grid in a separate window.

```
import random
import turtle

def roll_boggle_die(die):
    '''roll_boggle_die(die) -> str
    return a character'''
    letter = die[random.randrange(len(die))]
    return letter

def create_boggle_grid(boggledice):
    '''create_boggle_grid(boggledice) -> list
    returns a list that's a boggle grid'''
    diceList = boggledice[:]
    gridList = []
    while len(diceList) > 0:
        # pick a random die
        die = diceList[random.randrange(len(diceList))]
        # roll it and add it to the grid
        gridList.append(roll_boggle_die(die))
        # remove it from the list so we don't use it again
        diceList.remove(die)
    return gridList

def draw_boggle_grid(gridList,t):
    '''draw_boggle_grid(gridList,t) -> None
    uses turtle t to draw the boggle grid'''
    gridsize = 40
    for index in range(len(gridList)):
        t.goto((gridsize*(index%4-2),gridsize*(index//4+2)))
        letter = gridList[index]
        t.write(letter)
    t.goto((1000,0))  # move turtle out of the way

def convert_to_grid(num):
    '''convert_to_grid(num) -> (int,int)
    converts num 0-15 to grid coordinates'''
    return(num//4, num%4)
```

```
38
39 def adjacent_in_grid(pos1,pos2):
40     '''adjacent_in_grid(pos1,pos2) -> bool
41     returns True if they're adjacent'''
42     # convert to coordinates
43     coord1 = convert_to_grid(pos1)
44     coord2 = convert_to_grid(pos2)
45     if coord1==coord2:  # same square
46         return False
47     # both rows and columns must be at most 1 apart
48     return abs(coord1[0]-coord2[0])<=1 and abs(coord1[1]-coord2[1]
49
50 def letter_list(letter,gridList):
51     '''letter_list(letter,gridList) -> list
52     returns list of indices of letter in gridList'''
53     letterList = []
54     for index in range(len(gridList)):
55         if gridList[index] == letter:
56             letterList.append(index)
57     return letterList
58
59 def create_paths(listOfLists):
60     '''create_paths(listOfLists) -> list
61     returns a list of the paths of the lists in the argument
62     omits paths that go to the same point twice'''
63     paths = [[]]  # start with an empty path
64     for nextList in listOfLists:
65         newpaths = []
66         for priorPath in paths:
67             # extend each prior path by each number in the next li
68             for num in nextList:
69                 if num not in priorPath: # only extend if not a du
70                     newpaths.append(priorPath + [num])
71         paths = newpaths  # reset the current paths
72     return paths
73
74 def valid_word(word,gridList):
75     '''valid_word(word,gridList) -> bool
76     returns True if word is a valid word in gridList'''
77     letterLists = []
78     # make a list of where all the letters in word appear in the g
79     for char in word:
80         letterLists.append(letter_list(char,gridList))
81         if letterLists[-1] == []:  # letter doesn't exist
82             return False
83     # make all the possible paths of the letters
84     paths = create_paths(letterLists)
85     for path in paths:  # check if the path is a legal path
86         pathOK = True
87         for index in range(len(path)-1):
88             # make sure each letter is adjacent to the next letter
89             if not adjacent_in_grid(path[index],path[index+1]):
90                 pathOK = False
91                 break
92         if pathOK:
93             return True
94     return False
95
96 def play_boggle():
97     '''play_boggle() -> None
   plays a round of Boggle'''
```

```
 98  99      boggledice = [
100          'AAEEGN','ELRTTY','AOOTTW','ABBJOO',
101          'EHRTVW','CIMOTY','DISTTY','EIOSST',
102          'DELRVY','ACHOPS','HIMNQU','EEINSU',
103          'EEGHNW','AFFKPS','HLNNRZ','DELIRX'
104          ]
105      grid = create_boggle_grid(boggledice)
106      wordList = []
107      # create dictionary of legal words
108      wordFile = open('wordlist.txt','r')
109      legalWords = wordFile.readlines()
110      wordFile.close
111      # draw the grid using a turtle
112      wn = turtle.Screen()
113      alex = turtle.Turtle()
114      alex.penup()  # don't want any lines
115      draw_boggle_grid(grid,alex)
116      # get words from user
117      while True:
118          word = input("Enter your word (leave blank to quit): ").up
119          if word == '':
120              break  # done guessing
121          if word.lower()+'\n' not in legalWords:  # not a valid wor
122              print(word+' is NOT in the dictionary!')
123          elif valid_word(word,grid):
124              if len(word) < 3:  # too short
125                  print(word+' is not long enough.')
126              elif word in wordList:  # already found
127                  print('You already found '+word)
128              else:  # valid -- add to list
129                  print(word+' is a valid word!')
130                  wordList.append(word)
131          else:  # not in the grid
132              print(word+' is NOT in the grid!')
133      # compute score
134      print("Here's your score:")
135      score = 0
136      scoreChart = (0,0,0,1,1,2,3,5)
137      for word in wordList:
138          length = len(word)
139          if length >= 8:
140              wordScore = 11
141          else:
142              wordScore = scoreChart[length]
143          print(word+" scores "+str(wordScore))
144          score += wordScore
145      print("TOTAL SCORE: "+str(score))
146      print("Thanks for playing!")
147      # have computer look for longest word
148      print("Let me see...I bet I can find a long word.")
149      print("Please give me a moment to think...")
150      maxLength = 2
151      for word in legalWords:
152          word = word.strip('\n').upper()
153          if len(word) > maxLength and valid_word(word,grid):
154              longestWord = word
155              maxLength = len(word)
156      print("I found "+longestWord)
157      wn.mainloop()
158
play_boggle()
```

```
159
```

Run Pop Out ↗ Reset

Hint(s): Please feel free to ask for help or advice on the message board!

Your Response:
My program begins with the function getDictionary(), which reads the lines in a file and creates a new dictionary. The function puts each word in the dictionary with the word as both the key and value. It returns the dictionary. To do the optional part of the project, this function also creates a prefix dictionary - I will describe details of the prefix dictionary later.

Then, the function makeGrid() makes a grid by choosing a random die from the 16 Boggle dice and then picking a random side of that die. No die is reused. It then fills the first available blank space in the grid with the selected letter on the die. I keep the 16 Boggle dice in a list. Each member of the Boggle Dice list is itself a list - each with two members. If the first member (at 0th index) of the die is False, that means it has not been used yet. The second member contains all six letters on the die. If after randomly selecting a die from 16 dice, we find a die that has been used, we keep moving to the next die ((position + 1) modulo 16) until we find a die that has not yet been used. Once such an unused die has been found, we set the 0th element of the die to True to mark it as used and one of the sides of the die with a letter is randomly chosen. That letter is placed in the proper position in the frame of the grid, and eventually, the completed grid is returned.

With the grid and dictionary ready, we can now run the game like this:

While game is ON:
.....1. Print grid of letters to play with [printGrid()]
.....2. Get the player's input
.....3. If the input is not a blank:
.....If input is at least 3 letters long, has not already been successfully used and is a real word in the dictionary:
.....Find all positions in the grid where the first letter of the input appears
.....For each possible starting position in the grid of first letter of the word [findStartingPositions()]:
.....Mark the starting position as used by adding it to visitedLetters list
.....If rest of the letters of the word are in the grid [isInGrid()]:
.....Print appropriate message
.....Record score in list of scores
.....Else:
.....Print appropriate message
.....Else:
.....Print total score and all valid words entered by player with the corresponding score. Game is over.
End of the while loop

The fundamental piece of this program is the isInGrid() function. Its purpose is to figure out whether or not a word is present in the grid - it returns True when the whole word has been found or False if it has not. The main idea is to check all neighbors from the current position in grid and select some of the neighbors to recursively search for the next candidate letter. We only pick neighboring letters that we have not visited before and that match the next letter in the word that user typed. I used a list called visitedList to keep the visited positions [row, column] on the grid. This new visitedList is the only thing that changes in each recursive call. This list tells us two things:
1. the length of this list tells us which is the next letter in the word being checked
2. the last item of this list tells us the current position in the grid.
One important thing is that I had to make a new copy of the visitedList every time I was ready to recurse so that I would not modify the visitedList that was passed in to the current call of isInGrid().

isInGrid() uses a helper function called getMatchingPositions() that first generates a list of all neighbors. For example, it returns 3 neighbors in a list for a corner letter, but returns 8 neighbors for a center letter. From these neighbors, it selects the non-visited neighbors that have a matching letter and adds them to a list called matchingPositions, which is returned back to isInGrid().

Optional part of project: Finding the longest word in the grid:

I used most of my code from the isInGrid() function to do this part. As I described before, this function checks all neighbors of the current position in grid and selects some of these neighbors to recursively search. In the case of the Boggle game player, we only pick neighboring letters that match the next letter in the word that the user typed. In case of finding the longest word we have to match the next possible letter sequences that will create a valid word in the dictionary. We will have to build a special purpose dictionary for this that will contain ALL valid letter sequences for all words in the basic word dictionary. For example, if the word dictionary contains words BAM, BAN, BAT, BET and BEST, the letter sequence dictionary will contain word sequences stored in nested dictionaries with nesting level as deep as the longest word in the dictionary. It will have a structure like this:

```
B : { A : { M : {},
.............N : {},
.............T : {}
.............},
.....E : { S : { T : {}
.....................},
.............T : {}
...............}
.......}
```

Using this type of dictionary, we can check all neighbors of the current letter and select only the neighbors whose letter matches the letter sequence in the dictionary. We can keep recursing until we either find the end of a sequence in the dictionary or can find no usable neighbors in the grid.

We could write a simple minded search for this, without using this letter-sequence dictionary. However, this requires generating all possible word sequences within the grid. I think this would be incredibly slow and wasteful because this might require generating more than a billion candidate words, based on a rough estimate.

Testing:

I tested out my program in various ways. One of my games is shown below to show this. The grid follows:

```
H O R E
O E T I
E E O C
A U N T
```

I started out with various words that could be found on the grid such as ore, tore, teen, toe, aunt, cite, tire, hot, ton, tic, cot, hoe, rote, tone, cone, and noir.

Then, I tried out a word "cit", which I did not believe to be a word, to make sure my program was working correctly. As expected, the program asked me to enter another word. It did the same for the word "eeoc".

I tried out "tore", which had obviously been used already, and the program asked me to enter another word. Furthermore, I tried out "it", which was obviously too short (being less than 3 letters long), and the program asked me to enter another word.

I continued the game and ended with a total score of 22. The program also told me that the longest possible word to make in the grid was "tricot".

I played many other games like this on the Boggle Player to ensure the program worked. For example, sometimes, I checked to make sure I had not repeated a position that had already be used to create a word. If a grid had an "S" and an "I" on it, then I would not able to write out the word "sis". I performed similar tests on other cases and am fairly satisfied with the results. Here are some other test runs I did:

R A I I

I A H J

R U R V

A T B D

Enter your word (leave blank to quit):
Here's your score:
TOTAL SCORE: 0
Thanks for playing!
Let me see...I bet I can find a long word.
Please give me a moment to think...
I found URARI

Another test run with different types of player mistakes I checked for:
>>>
O W P O

M U W N

I S R L

L R T T

Enter your word (leave blank to quit): mom
MOM is NOT in grid!

O W P O

M U W N

I S R L

L R T T

Enter your word (leave blank to quit): pow
POW is a valid word!

O W P O

M U W N

I S R L

L R T T

```
Enter your word (leave blank to quit): zzbb
ZZBB is NOT in the dictionary!

O W P O

M U W N

I S R L

L R T T

Enter your word (leave blank to quit): sum
SUM is a valid word!

O W P O

M U W N

I S R L

L R T T

Enter your word (leave blank to quit): sum
SUM has already been used!

O W P O

M U W N

I S R L

L R T T

Enter your word (leave blank to quit): no
NO is NOT at least 3 letters long!

O W P O

M U W N

I S R L

L R T T

Enter your word (leave blank to quit):
Here's your score:
POW scores 1
SUM scores 1
TOTAL SCORE: 2
Thanks for playing!
Let me see...I bet I can find a long word.
Please give me a moment to think...
I found STRUM
```

Your Code:

```
# Python Class 926
# Lesson 11 Problem 1
# Author: suchita (216038)

# Project #3: Boggle Game Player

def get_dictionary(prefixDict):
    '''get_dictionary() -> dict
    Returns dict with each word in the dictionary file
    as both a key and a value
    Creates a prefix based dictionaries by breaking each word into letters'''
    # read each line and make a dictionary so that later we
    # can check whether or not the player's input is a word
    inputFile = open('wordlist.txt','r')
    wordList = inputFile.readlines()
    inputFile.close()
    # create a dictionary
    wordDict = {}
    for word in wordList:
        newWord = word.strip().upper()
        # put each word in the dictionary
        wordDict[newWord] = newWord
        tempDict = prefixDict
        # add each letter in the word to the prefix based dictionary
        for letter in newWord:
            # get the value of each letter or give it the value of an empty
            # dictionary if the letter is not in the dictionary
            tempDict[letter] = tempDict.get(letter, {})
            # tempDict is reassigned to the value of the previous tempDict
            # so that we can build a chain of dictionaries using the
            # letters in the word
            tempDict = tempDict.get(letter)
    return wordDict

def make_grid():
    '''make_grid() -> None
    Makes a 4x4 random grid from the sides of the 16 6-sides dice.'''
    import random
    # sides of dice are in a list with word "False" to show
    # a side of the die has not been placed in the grid yet -
    # when the die is used, "False" is changed to "True"
    dice = [[False, 'AAEEGN'],
            [False, 'ELRTTY'],
            [False, 'AOOTTW'],
            [False, 'ABBJOO'],
            [False, 'EHRTVW'],
            [False, 'CIMOTU'],
            [False, 'DISTTY'],
            [False, 'EIOSST'],
            [False, 'DELRVY'],
            [False, 'ACHOPS'],
            [False, 'HIMNQU'],
            [False, 'EEINSU'],
            [False, 'EEGHNW'],
            [False, 'AFFKPS'],
            [False, 'HLNNRZ'],
            [False, 'DELIRX']]

    # represent basic frame of the 4x4 grid
    grid = [[' ', ' ', ' ', ' '],
            [' ', ' ', ' ', ' '],
            [' ', ' ', ' ', ' '],
            [' ', ' ', ' ', ' ']]

    # for each space in grid
    for row in range(4):
        for column in range(4):
            # choose one of the dice
            randomDie = random.randrange(16)
            die = dice[randomDie] # plural: dice, singular: die
            # move forward till we find an unused die
            while die[0] == True:
                randomDie = (randomDie + 1) % 16
```

```
74                  die = dice[randomDie]
75              die[0] == True # mark the die as used
76              # choose one of the sides on the die
77              randNum = random.randrange(6)
78              # place the letter on the side onto the grid
79              grid[row][column] = die[1][randNum]
80      return grid
81
82  def print_grid(grid):
83      '''print_grid(list) -> None
84      Prints a matrix-like structure with 4 rows and 4 columns'''
85      # for each space in the grid
86      for row in range(4):
87          for column in range(4):
88              # create grid to show player
89              print(grid[row][column] + "  ",end='')
90          print('\n')
91      return
92
93  def copy_list(oldList):
94      '''copy_list(list) -> list
95      Returns a copy of the previous list'''
96      newList = []
97      # make a new list to change
98      for item in oldList:
99          newList.append(item)
100     return newList
101
102 def get_neighbor_positions(position):
103     '''get_neighbor_positions(list) -> list
104     Returns a list of positions where a surrounding letter can be'''
105     # 8 places for surrounding letters to be
106     row = position[0]
107     column = position[1]
108     # these are the possible positions where the neighboring letter can be
109     possibleNeighborPositions = [[row - 1, column - 1], [row - 1, column],
110                          [row - 1, column + 1], [row, column - 1],
111                          [row, column + 1], [row + 1, column - 1],
112                          [row + 1, column], [row + 1, column + 1]]
113     neighborPositions = []
114     # go through each possible position
115     for positions in possibleNeighborPositions:
116         # some spaces are not surrounded by letters on 8 sides
117         if positions[0] > -1 and positions[0] < 4:
118             if positions[1] > -1 and positions[1] < 4:
119                 # the position has a letter in it
120                 neighborPositions.append(positions)
121     return neighborPositions
122
123 def get_matching_positions(grid, currentPosition, letter, visitedLetters):
124     '''get_matching_positions(list, list, string, list) -> list
125     Returns a list of positions where letter is when only
126     looking at the positions surrounding the previous letter'''
127     neighbors = get_neighbor_positions(currentPosition)
128     matchingPositions = []
129     # go through each position
130     for position in neighbors:
131         if position not in visitedLetters:
132             # find out what the row and column of the position is
133             row = position[0]
134             column = position[1]
135             # check if the letter is in this position
136             if grid[row][column] == letter:
137                 matchingPositions.append(position)
138     return matchingPositions
139
140 def is_in_grid(word, grid, visitedLetters):
141     '''is_in_grid(string, list, list) -> boolean
142     Returns True if word is in grid and False otherwise'''
143     # stop when the whole word has been found
144     if len(word) == len(visitedLetters):
145         return True
146     # this is how many letters in the word we have gone through
147     letterIndex = len(visitedLetters)
148     # the computer starts counting at 0
149     nextLetter = word[letterIndex]
```

```
150     # this is where we are now
151     currentPosition = visitedLetters[letterIndex - 1]
152     # find possible places for letter to be around the letter
153     matchingPositions = get_matching_positions(grid, currentPosition,
154                                                nextLetter, visitedLetters)
155     # cannot find letter in grid
156     if len(matchingPositions) == 0:
157         return False
158     for position in matchingPositions:
159         # we do not want to change the original list
160         newVisitedLetters = copy_list(visitedLetters)
161         newVisitedLetters.append(position)
162         # we do not need to find another of the same word
163         if is_in_grid(word, grid, newVisitedLetters):
164             return True
165     return False
166 
167 def find_starting_positions(letter, grid):
168     '''find_starting_positions(string, list) -> list
169     Returns a list of positions where a letter occurs in the grid'''
170     newList = []
171     # go through each space in grid
172     for row in range(4):
173         for column in range(4):
174             # check if the letter is in  the position
175             if grid[row][column] == letter:
176                 newList.append([row, column])
177     return newList
178 
179 def is_not_already_used(word, scoreList):
180     '''is_not_already_used(string, list) -> boolean
181     Returns True if word has not been used and False otherwise'''
182     for item in scoreList:
183         # we used the word already
184         if word == item[0]:
185             return False
186     return True
187 
188 def get_score(word):
189     '''get_score(string) -> int
190     Returns score for a word'''
191     # score depends on length
192     length = len(word)
193     if length == 3 or length == 4:
194         score = 1
195     elif length == 5:
196         score = 2
197     elif length == 6:
198         score = 3
199     elif length == 7:
200         score = 4
201     else:
202         score = 6
203     # value of each word is returned
204     return score
205 
206 def print_scores(scoreList):
207     '''print_scores(list) -> None
208     Prints score for each word and prints total score'''
209     print("Here's your score:")
210     # print the value of each word
211     for item in scoreList:
212         print(item[0] + " scores " + str(item[1]))
213     totalScore = 0
214     # find the total of the scores
215     for item in scoreList:
216         totalScore += item[1]
217     print("TOTAL SCORE: " + str(totalScore))
218     print("Thanks for playing!")
219 
220 def get_matching_prefixes(prefixDict, position, grid, visitedLetters):
221     '''get_matching_prefixes(dict, list, list, list) -> list
222     Returns all matching positions in dictionary'''
223     neighbors = get_neighbor_positions(position)
224     matchingPositions = []
225     # go through each position
```

```
226     for position in neighbors:
227         if position not in visitedLetters:
228             # find out what the row and column of the position is
229             row = position[0]
230             column = position[1]
231             # check if the letter is in this position
232             if grid[row][column] in prefixDict:
233                 # add this position to list
234                 matchingPositions.append(position)
235     return matchingPositions
236 
237 def word_from_visited_letters(grid, visitedLetters):
238     '''word_from_visited_letters(list, list) -> string
239     Returns word from positions of visited letters'''
240     # we will concatenate various letters to this
241     word = ''
242     for position in visitedLetters:
243         r = position[0]
244         c = position[1]
245         # letters are joined together into word
246         word = word + grid[r][c]
247     return word
248 
249 def find_longest_word(prefixDict, dictionary, grid, visitedLetters,
250                       longestWord):
251     '''find_longest_word(dict, dict, list, list, string) -> string
252     Returns longest word one can make from a certain starting point
253     Most of the code is based on is_in_grid()'''
254     # stop when you have reached the current dictionary path
255     if len(prefixDict) == 0:
256         return longestWord
257 
258     # this is how many letters in the word we have gone through
259     letterIndex = len(visitedLetters)
260     # this is where we are now
261     currentPosition = visitedLetters[letterIndex - 1]
262     # find possible places for letter to be around the letter
263     matchingPositions = get_matching_prefixes(prefixDict, currentPosition,
264                                               grid, visitedLetters)
265 
266     # cannot find letter from the prefix dictionary in the grid
267     if len(matchingPositions) == 0:
268         return longestWord
269 
270     for position in matchingPositions:
271         # we do not want to change the original list
272         newVisitedLetters = copy_list(visitedLetters)
273         newVisitedLetters.append(position)
274         # form a word from newVisitedLetters
275         word = word_from_visited_letters(grid, newVisitedLetters)
276         if len(word) > len(longestWord) and word in dictionary:
277             #print(word + ' replaces ' + longestWord + ' as longest')
278             # word replaces longestWord as longest
279             longestWord = word
280         currentLetter = word[len(word) - 1]
281         #print(word)
282         # we want to find a longer word
283         returnedWord = find_longest_word(prefixDict[currentLetter], dictionary,
284                                          grid, newVisitedLetters, longestWord)
285         if len(returnedWord) > len(longestWord) and returnedWord in dictionary:
286             #print(returnedWord + ' replaces ' + longestWord + ' as longest')
287             # returnedWord replpaces longestWord as longest
288             longestWord = returnedWord
289 
290     return longestWord
291 
292 def print_longest_word(grid, dictionary, prefixDict):
293     '''print_longest_word(list, dict) -> string
294     Prints message and longest word in grid'''
295     print("Let me see...I bet I can find a long word.")
296     print("Please give me a moment to think...")
297     longestWord = ''
298     # going through each position in the grid
299     for row in range(4):
300         for column in range(4):
301             position = [row, column]
```

```
            visitedLetters = [position]
            # find the longest word when starting from position
            word = find_longest_word(prefixDict, dictionary, grid,
                                     visitedLetters, longestWord)
            if len(word) > len(longestWord) and word in dictionary:
                #print(word + ' replaces ' + longestWord + ' as longest')
                # word replaces longestWord as longest
                longestWord = word
    print('I found ' + longestWord)

def play_game(wordDict, grid, prefixDict):
    '''play_game(dict, list, dict) -> None
    Plays boggle until player inputs a blank
    Prints a grid and asks player to enter word on grid
    Print various responses depending on validity of word given by player
    Print scores and game ends
    Prints longest possible word to create'''
    done = False
    # we do not have any scores yet
    scores = []
    # when the game is running
    while done == False:
        print_grid(grid)
        # letters are made uppercase to make everything easier
        playerInput = input("Enter your word (leave blank to quit): ").upper()
        # if game is not stopped
        if playerInput != '' and playerInput != ' ':
            # word is long enough
            if len(playerInput) >= 3:
                # word has not been used
                if is_not_already_used(playerInput, scores):
                    # word is a real word
                    if playerInput in wordDict:
                        # find out where first letter is
                        positions = find_starting_positions(playerInput[0],
                                                            grid)
                        foundInGrid = False
                        # go through each possible starting point
                        for position in positions:
                            if not foundInGrid:
                                # we have checked this position
                                visitedLetters = [position]
                                if is_in_grid(playerInput, grid,
                                              visitedLetters):
                                    # word can be found in grid
                                    foundInGrid = True
                                    print(playerInput + " is a valid word!\n")
                                    # record score
                                    score = get_score(playerInput)
                                    scores.append([playerInput, score])
                        # word is not in grid
                        if not foundInGrid:
                            print(playerInput + " is NOT in grid!\n")
                    # word is not a real word
                    else:
                        print(playerInput + " is NOT in the dictionary!\n")
                # word has been used
                else:
                    print(playerInput + " has already been used!\n")
            # word is not long enough
            else:
                print(playerInput + " is NOT at least 3 letters long!\n")
        # game is over
        else:
            # print scores
            print_scores(scores)
            print_longest_word(grid, dictionary, prefixDict)
            # stop while loop
            done = True
    return

prefixDict = {}
dictionary = get_dictionary(prefixDict)
grid = make_grid()
# play boggle
play_game(dictionary, grid, prefixDict)
```

Run Pop Out ↗ Reset

Technical Score: 7 / 7
Style Score: 1 / 1
Comments:
Great job! Your game works very well, save for one very small error with the score values. You have a great description along with your game, and that makes it much easier to understand which functions perform which tasks.

Though your method for randomizing the dice is valid, it's possible for the method to take quite a while. If there's only one dice left to roll, you'll have to keep doing `random.randrange(16)` until you find that one single dice, and this might take a while! Instead, take a look at the `random.shuffle()` function. It shuffles all the elements of the list passed in as an argument. For example:

```
import random
# Your dice variable
dice = [[False, 'AAEEGN'],
            [False, 'ELRTTY'],
            [False, 'AOOTTW'],
            [False, 'ABBJOO'],
            [False, 'EHRTVW'],
            [False, 'CIMOTU'],
            [False, 'DISTTY'],
            [False, 'EIOSST'],
            [False, 'DELRVY'],
            [False, 'ACHOPS'],
            [False, 'HIMNQU'],
            [False, 'EEINSU'],
            [False, 'EEGHNW'],
            [False, 'AFFKPS'],
            [False, 'HLNNRZ'],
            [False, 'DELIRX']]
random.shuffle(dice)
print(dice)
```

Run Pop Out ↗ Reset

If you click the run button above, you'll notice that the list is shuffled. This shuffles your dice list more reliably and quicker than your current algorithm!

Your method for finding the longest word seems rather complicated. It's clever, and it works, but there's a far simpler way! What if you could order the `wordlist.txt` dictionary by length? If you're interested, take a look at the `.sort()` function. It takes in a few optional arguments, one of which (`key`) tells the function how to sort. For example:

```
# Your code
inputFile = open('wordlist.txt', 'r')
wordList = inputFile.readlines()
inputFile.close()

wordList.sort(key = len, reverse = true) #Sort by length, reverse
```

Run Pop Out ↗ Reset

(The code above doesn't really run because `wordlist.txt` doesn't exist here.)

`reverse = true` is needed because otherwise, the words would be sorted from shortest to longest. Then, you could iterate over this wordList, checking each word to see if it's contained in your board. The first word that you find should be the longest word (because it's ordered longest to shortest).

While we're on the topic of finding the longest word, your code finds just one longest word. What if there are two words with the same length? Is it possible to find both of those words? What if there are more? See if you can find them all!

By the way, the score values you use in `get_score()` don't quite match the scores given in the problem prompt. Make sure you don't miss the small details!

You do a great job with splitting your code into smaller functions! One of the primary advantages to this is that you can find bugs or test your code much more easily. For example, you can test one specific function to make sure it works. You should try showing a few tests in which you test one of your functions rather than running your game over over! You could make a very simplified board and a simplified dictionary, and show that your `find_longest_word()` function works as you wish.

You have thanked the grader!