Lab 10: Functions

Mar 1, 2018

Main Event

1. Temperature conversion

Write two functions that convert Fahrenheit to Celsius, and Celsius to Fahrenheit, respectively. Each function should take a single argument—the temperature—and return the converted value.

Solution:

```
# Fahrenheit to Celsius
def f_to_c(fahrenheit):
    return (fahrenheit - 32) / (9 / 5)

# Celsius to Fahrenheit
def c_to_f(celsius):
    return celsius * 9 / 5 + 32
```

2. List reversal

Write a function that reverses a list. The function should take a list and return a reversed version. The original list—the one that the user has provided—should not be altered!

Solution:

```
def list_reverse(lst):
    rev = []
    for i in lst:
        rev.insert(0, i)

    return rev
```

3. Tic-Tac-Toe

Tic-tac-toe is a grid-based game between two players in which the objective is for one player to cover three consecutive vertical, horizontal, or diagonal tiles. The game can be developed in three steps: asking the user for a coordinate, placing that either an ‘x’ or ‘y’ at that coordinate (depending on the player), and determining if the placed coordinate creates a winner:

1. Write a function that creates an empty board. The dimensions of the board should be specified as arguments:

   ```
   def make_board(m, n, filler):
   ```

   This function should basically return a list-of-lists: a list containing *m* lists of *n* elements. Elements should have the value `filler`.

2. Write a function that prints a board. The function should take an *m*-by-*n* matrix and print the contents of that matrix using the ASCII decorators that we’ve previously used:

   ```
   def print_board(board):
   ```

   In practice, we should be able to do the following:

   ```
   >>> board = make_board(3, 3, ' ')
   >>> print_board(board)
   +---+---+---+
   |   |   |   |
   +---+---+---+
   |   |   |   |
   +---+---+---+
   |   |   |   |
   +---+---+---+
   ```

   You can assume that all values in the board will be a single character or digit. Note, however, that you cannot assume the values will be strings—your function should work regardless.

3. Write a function that updates the board:

   ```
   def update_board(board, i, j, decorator, filler):
   ```

where `board` is the board, `i` and `j` are the coordinates, and `decorator` is the character you want to add; generally either ‘x’ or ‘o’, depending on the player. This function should return true or false depending on whether the board was actually updated—remember, in tic-tac-toe a player can only mark spots that are “open” (equal-to `filler`).

4. Write a series of functions that check to see if there is a winner. In tic-tac-toe there is a winner if a player takes an entire row, column, or diagonal—you should write functions for each:

   ```
   def is_horizontal_winner(board, decorator)
   def is_vertical_winner(board, decorator)
   def is_diagonal_winner(board, decorator)
   ```

   In each case, `board` is the board and `decorator` is the character for which we want to check. Each function should return a Boolean value signifying whether there is a winner in its respective case. For purposes of this assignment, you can assume all boards will have the same number of rows and columns.

   Finally, compose these functions into a single function:

   ```
   def iswinner(board, decorator)
   ```

   that returns the disjunction of the previous three.

Solution:

```
# create the empty board
def make_board(dim, filler):
    board = []
    for i in range(dim):
        inner = []
        for j in range(dim):
            inner.append(filler)
        board.append(inner)

    return board

# print the board
def print_board(board):
    i = 0
```

```python
    for i in range(len(board)):
        row = board[i]
        print(' ', ' | '.join(row), ' ')
        if i < len(board) - 1:
            print('+---' * len(row) + '+')

# update the board
def update_board(board, x, y, decorator, filler):
    legal = board[x][y] == filler
    if legal:
        board[x][y] = decorator

    return legal

# Given a sequence, determine whether all elements in that sequence
# are the decorator. This function was not required. It was written tc
# programming the other is_*_winner functions easier.
def consecutive(block, decorator):
    sentence = ''.join(block)
    remaining = sentence.strip(decorator)

    return not remaining

# vertical winner
def is_vertical_winner(board, decorator):
    for row in board:
        if consecutive(row, decorator):
            return True

    return False

# horizontal winner
def is_horizontal_winner(board, decorator):
    columns = len(board) # assume the board is square

    for i in range(columns):
        col = []
        for j in range(columns):
            col.append(board[j][i])
        if consecutive(col, decorator):
            return True

    return False
```

```
# diagonal winner
def is_diagonal_winner(board, decorator):
    columns = len(board)
    ldiag = []
    rdiag = []

    for i in range(columns):
        ldiag.append(board[i][i])
        rdiag.append(board[i][-i-1])

    return consecutive(ldiag, decorator) or consecutive(rdiag, decorat

def iswinner(board, decorator):
    return (is_vertical_winner(board, decorator) or
            is_horizontal_winner(board, decorator) or
            is_diagonal_winner(board, decorator))
```

4. Putting it all together

Now that you have all the functions, you can build the game. First create the board and print it. Next, within a loop, ask a player for their input, print the board, and check to see if there is a winner. If there is, end the game; otherwise continue on to the next player (you can assume two players). This portion does not have to be in a function.

Solution:

```
# assume the functions developed in Exercise 3 are in a file called
# ttt.py
import ttt

# convenience function for printing the board
def display(board):
    print()
    ttt.print_board(board)
    print()

# variables used throughout game play
dimensions = 3
filler = ' '
markers = [ 'x', 'o' ]
```

```python
player = 0

# initialise the game: create the board and print it
board = ttt.make_board(dimensions, filler)
display(board)

# play!
level = 0
while level < dimensions ** 2:
    # whose turn?
    decorator = markers[level % len(markers)]

    # get the user input
    prompt = decorator + ' enter a position [row,col]: '
    coordinates = input(prompt).split(',')
    row = int(coordinates[0])
    col = int(coordinates[1])

    # update the board
    updated = ttt.update_board(board, row, col, decorator, filler)
    if not updated:
        print('Illegal move! Try again.')
        continue

    # print the board
    display(board)

    # determine if the update created a winner
    if ttt.iswinner(board, decorator):
        print(decorator, 'wins!')
        break

    level += 1
```

Additional Practice

1. Identifying ties

One of the things our tic-tac-toe game did not do was determine if the game was a tie. A tie in tic-tac-toe is when there are no more available spaces for users to place their mark. This exercise asks you to update your code to do so. Specifically, write a function

```
def istie(board, filler)
```

that returns `True` or `False` depending on whether the board is in a tied state. Integrate this function into your existing code so that game play is stopped if this is the case.

Solution:

```
def istie(board, filler):
    for row in board:
        for decorator in row:
            if decorator == filler:
                return False

    return True
```

2. Generalized version

We previously assumed a traditional version of tic-tac-toe in which the board was a 3-by-3 grid. Update your game such that it can be played on a board of any size. In cases where the board does not present a corner-to-corner diagonal, you are free to define what being a diagonal winner actually means.