EN.540.635 "Software Carpentry"

Problem Solving in Python

Introduction

In this handout I have compiled 5 example problems from one of the resources listed for the course: Leetcode. You can look up the corresponding problem description/statement on the Leetcode website, accessing it through the provided hyperlink section titles once you have made an account on there. You are welcome to attempt these problem and others listed on Leetcode yourself to become more 'fluent' and confident in Python. Please note that I have not added documentation for the functions that I used to solve the problems, but starting some point in the class you will be expected to for the homework assignments.

Shortest Distance to a Character

```python
def shortestToChar(S, C):
    dist = []
    # list comprehension to store indices at which
    # the letter C is present in string S
    # Go through the string and store the index
    # i that you get from the enumerate generator
    # whenever the current letter is the same as C
    c_pos = [i for i, c in enumerate(S) if c == C]
    # looping through only the indices and calculating
    # the shortest absolute difference b/w all the positions
    # and the list of stored char positions. The distances are
    # appended to dist
    for i in range(len(S)):
        c_dist = [abs(i - c) for c in c_pos]
        dist.append(min(c_dist))

    return dist


if __name__ == '__main__':
    print(shortestToChar('loveleetcode', 'e'))
```

Rotated Digits

```python
def rotatedDigits(N):
    # dictionary to store what different digits
    # become when rotated
    digit_dict = {
        '0': '0', '1': '1', '2': '5',
        '5': '2', '6': '9', '8': '8', '9': '6'
    }
    # increment by 1 whenever the 'good number'
    # conditions are met
    good_count = 0
    for i in range(1, N + 1):
        num = str(i)
        # keep adding the flipped digits
        new_num = ''
```

```
        for dig in num:
            if dig not in digit_dict:
                # break out of the loop if any
                # digits that are flippable are
                # encountered
                valid = False
                break
            else:
                new_num += digit_dict[dig]
                valid = True
        # the number is only 'good'
        # if all the digits are flippable and
        # the new number is different from the OG
        if new_num != num and valid:
            good_count += 1
    return good_count


if __name__ == '__main__':
    print(rotatedDigits(30))
```

ZigZag Conversion

```
def convert(inp_str, numRows):
    if numRows <= 1:
        return inp_str
    else:
        # create list of empty strings correpsonding
        # to each row of the new zigzag string
        new_str = ['' for r in range(numRows)]
        # keep track of the row that you want to add
        # the letter to in the new string
        row_count = 0
        # the direction in which to travel the rows
        # 1 means you are traveling down
        # -1 means you are traveling up
        add_row = 1
        for s in inp_str:
            new_str[row_count] += s
            # travel down or up in the new str
            row_count = row_count + add_row
            # 2 lettered strings will come out the same
            if len(inp_str) > 2:
                # if you either reach the botton row after traveling
                # down or reach the top row after traveling up
                # switch the directions
                if any([row_count == numRows - 1, not row_count]):
                    add_row = add_row * -1
        # join the different list of strings in
        # new_str to form 1 continuos string
        new_str = ''.join(new_str)

        return new_str
```

```
if __name__ == '__main__':
    print(convert("PAYPALISHIRING"))
```

Validate IP address

```
import string


def validIPAddress(IP):
    # IPv4 addresses have a fullstop in the
    if '.' in IP:
        # split the address along the full stop
        # this gives us a list of strings
        ip_list = IP.split('.')
        # make sure each string is not empty
        # and that we have only 4 components in
        # in the address
        str_lens = [len(num) > 0 for num in ip_list]
        if len(ip_list) == 4 and all(str_lens):
            # for each string component
            # checking if the first letter is 0
            # only if the string is greater that 1 letter
            leading_zeroes = [
                num[0] == '0' and len(num) > 1 for num in ip_list
            ]
            # if all the strings do not have leading zeroes
            if not any(leading_zeroes):
                # make sure that all strings are numeric
                is_num = [num.isnumeric() for num in ip_list]
                if all(is_num):
                    # convert all string components to integers
                    num_list = list(map(int, ip_list))
                    # apply the number range constraints on all
                    # the numbers
                    num_range = [0 <= num <= 255 for num in num_list]
                    if all(num_range):
                        return 'IPv4'
                    else:
                        return 'Neither'
                else:
                    return 'Neither'
            else:
                return 'Neither'
    # IPv6 addresses have colons in them
    elif ':' in IP:
        # split laong the colon
        ip_list = IP.split(':')
        if len(ip_list) == 8:
            # check if each split component has between 1
            # and 4 characters
            str_lens = [1 <= len(num) <= 4 for num in ip_list]
```

```
            if all(str_lens):
                # check if different string components are valid
                # hexadecimal numbers
                hex_valid = [
                    c in string.hexdigits
                    for num in ip_list
                    for c in num
                ]
                if all(hex_valid):
                    return 'IPv6'
                else:
                    return 'Neither'
            else:
                return 'Neither'
        else:
            return 'Neither'
    else:
        return 'Neither'

    return 'Neither'


if __name__ == '__main__':
    print(validIPAddress("02001:0db8:85a3:0000:0000:8a2e:0370:7334"))
```

Magic Squares in Grid

```
def check_magic_sum(sub_grid):
    # calculate and store the sums of the
    # 2 diagonal in a 3 x 3 grid
    diag2_sum = 0
    diag1_sum = 0
    for r in range(3):
        # get the elements of the rth column
        curr_col = [row[r] for row in sub_grid]
        # check sum of rth row
        if not sum(sub_grid[r]) == 15:
            return False
        # check sum of rth columns
        elif not sum(curr_col) == 15:
            return False
        # add the diagonal element r, r to the sum
        diag1_sum += sub_grid[r][r]
        # add the diagonal element (2 - r), (2 - r)
        # to the sum
        diag2_sum += sub_grid[2 - r][2 - r]
    # check if both diagonal sums are 15
    if diag1_sum != 15 or diag2_sum != 15:
        return False

    return True
```

```
def numMagicSquaresInside(grid):
    # count the number of valid magic sub squares
    magic_count = 0
    num_rows = len(grid)
    num_cols = len(grid[0])
    for r in range(num_rows):
        for c in range(num_cols):
            # get the ending value for the
            # row and columns starting at coordinate r, c
            r_lim = r + 3
            c_lim = c + 3
            # check if the end points are within the
            # original grid
            if r_lim > num_rows or c_lim > num_cols:
                pass
            else:
                # get the rows in between r and r_lim
                # and then the elements b/w c and c_lim
                sub_grid = [row[c: c_lim] for row in grid[r: r_lim]]
                # flatten the 2D list into a linear list
                flat_list = [c for row in sub_grid for c in row]
                # get a set of the flat list to remove
                # duplicate elements
                flat_set = set(flat_list)
                # if there are no duplicate elements
                # both flate_set and flat_list
                # will have the same length
                if len(flat_list) == len(flat_set):
                    # check if all the elements are distinct
                    num_range = [0 < num < 10 for num in flat_list]
                    # check if the subsquare us magic or not
                    if check_magic_sum(sub_grid) and all(num_range):
                        magic_count += 1

    return magic_count


if __name__ == '__main__':

    inp_sqr = [
        [4, 3, 8, 4],
        [9, 5, 1, 9],
        [2, 7, 6, 2]
    ]

    print(check_magic_sum(magic_square))
```