Introduction to Programming

Lab 3: Connect Four

Fatmanur Gençer, Christian Callau

2018-19 First Trimester

0. Introduction

Connect is a two-player connection game in which the players first choose a color and then take turns dropping one colored disc from the top into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the lowest available space within the column. The objective of the game is to be the first to form a horizontal, vertical, or diagonal line of four of one's own discs.

1. Board info disposition

We will store the board info in a N x M matrix. With a width of 7 columns and a height of 6 rows. We will fill the matrix with integer values. 2 will signify empty slot; 0 will signify a player 0 chip and 1 a player 1 chip. The position 0, 0 of the matrix will map to the left-most bottom-most position of the visual representation of the board. And the position 5, 6 to the right-most top-most position.

For simulating the gravity fall of each chip we will create an additional list that will store an index for each column, this index will point to the next free spot of the column starting with the bottom-most position (0) and incrementing each time a chip is played in that column until it reaches the top (5), at this point checking if the index is greater or equal than the height will return us if the column is full.

2. Player control and color

We choose color Yellow for player 0 and color Red for player 1. The player code will be an integer swapping between the integer values 0 and 1. For swapping we can do it very easy in one line using the xor operator. For storing the colors, we will make a tuple and the player value will correspond to the index.

3. Checking 4-on-line

When playing a move, we must check if there is a 4-on-line combination. There are multiple ways of doing that. The first approach it comes to mind is iterating over the whole matrix checking for the pattern horizontally, vertically and in both diagonal directions. This approach would work fine but we can do better. When playing a new chip there is only one spot where a 4-on-line combination can appear and that is at the same row, column, diagonal and counter diagonal where the chip has been

placed. So, we only must check these places and the algorithm ends up being more efficient. We won't notice in this board size but if the board would have an arbitrary N x M dimension we would.

So, we start by defining methods that given a matrix and a r and c coordinates returns us the row, the column, the diagonal and the counter diagonal. We start with the easy ones.

For getting the row we got it easy, all rows are already stored in order in the matrix, we just must return the row corresponding to the r coordinate.

```
1.  # Returns the row of the matrix at the position r and c
2.  def get_row(mat, r, c):
3.    return mat[r]
```

For getting the column is almost as straight forward as before, but in this case, we must construct a new list, with a list comprehension we can do it in one line, by iterating over all the rows and getting the element corresponding to the column c.

```
1.  # Returns the column of the matrix at the position r and c
2.  def get_col(mat, r, c):
3.    return [row[c] for row in mat]
```

Getting the diagonal is a bit trickier. We noticed a pattern that is if we subtract the x and y components of each coordinate the result is the same for the whole diagonal, and each diagonal has an own result that is different from the adjacent diagonals by 1. If we run this simple code:

```
1.  rows = 6
2.  cols = 7
3.  matx = [['{: d}'.format(col - row) for col in range(cols)] for row in range(rows)]
4.
5.  print()
6.  for row in matx:
7.    print('  ' + ' '.join(row))
```

We can see the result:

```
1.  C:\Users\X3\Desktop>python test.py
2.
3.     0  1  2  3  4  5  6
4.    -1  0  1  2  3  4  5
5.    -2 -1  0  1  2  3  4
6.    -3 -2 -1  0  1  2  3
7.    -4 -3 -2 -1  0  1  2
8.    -5 -4 -3 -2 -1  0  1
```

This value could be very useful when iterating over the matrix for generating a list that contains all the elements of the diagonal. For making it easier to visualize we make a simple modification to the previous code to print the same matrix with the row and col coordinates of each position, and this is the result:

```
1.  C:\Users\X3\Desktop>python test.py
2.
3.    00 01 02 03 04 05 06
4.    10 11 12 13 14 15 16
5.    20 21 22 23 24 25 26
6.    30 31 32 33 34 35 36
7.    40 41 42 43 44 45 46
8.    50 51 52 53 54 55 56
```

We quickly see that if we iterate over the height of the matrix and we use the same index for the row and col coordinates we are going through the main diagonal.

It is the same for the top adjacent diagonal with the difference that the column coordinates it has an increment of 1 from the previous case. We see the same with the next diagonal with a +2 for the column. We see a clear relation with this and the values calculated in the first picture. We can assign the result of subtracting (col - row) to a variable called offset and then add the offset to the column coordinate.

But there is one problem, the diagonals below the main diagonal seems have a different behaviour, is the row coordinate that is shifted instead of the column? How do we fit the previous solution to this case? The answer is that the previous solution it will work the same if we imagine the matrix to have more cells at the left. Those cells will have negative column coordinates if we stick the position 0, 0 as a reference at the same point as before:

```
C:\Users\X3\Desktop>python test.py

   0-4 0-3 0-2 0-1 00 01 02 03 04 05 06
   1-4 1-3 1-2 1-1 10 11 12 13 14 15 16
   2-4 2-3 2-2 2-1 20 21 22 23 24 25 26
   3-4 3-3 3-2 3-1 30 31 32 33 34 35 36
   4-4 4-3 4-2 4-1 40 41 42 43 44 45 46
   5-4 5-3 5-2 5-1 50 51 52 53 54 55 56
```

There we can see that if we choose for instance the coordinates (4, 0) and we start iterating from 0 to form the diagonal, the offset will be equal to -4, and adding this offset to the row coordinates we will get the sequence (0, -4), (1, -3), (2, -2), (3, -1), (4, 0), (5, 1). Which is exactly the diagonal of the position (4, 0) and it fits well with the general solution we came up. The only thing we are missing is to limit the coordinates to fit into the matrix. For that we only must check that the column + offset is between [0, with). We can finally generate the list containing the elements of the matrix at a given coordinates in one line with a list comprehension with an if conditional to check the bounds like this:

```
# Returns the diagonal of the matrix at the position r and c
def get_dig(mat, r, c):
  off = c - r
  return [row[i+off] for i, row in enumerate(mat) if 0 <= i+off < len(row)]
```

For getting the counter diagonal we thought that a similar approach could be good. But after some time trying to find a pattern, that by the way we found, but later fitting it to the solution proved to be a bit more complex so we started to search for other approach and then we thought that we could reuse the same function we just made to get the diagonal. The only additional thing we must do is flip the matrix horizontally, getting the diagonal of the matrix flipped like that is the same as the counter diagonal, in addition we must flip the column coordinate too. So that's what we did and works nicely:

```
# Returns the counter diagonal of the matrix at the position r and c
def get_cdg(mat, r, c):
  flip_horizontal_mat = [row[::-1] for row in mat]
  new_c = len(mat[0])-1 - c
  return get_dig(flip_horizontal_mat, r, new_c)
```

Once we have the list which the elements of each of the four directions we just must check if there is a combination of 4-on-line 0s or 1s in that list. We could do it by iterating over the list and comparing the first element, the second, the third and the fourth with 0 or 1, and then incrementing the offset

of each position to then compare the second, third, fourth and five, and so one until the last element of the list is one of the elements we are comparing. But there is a shorter, easiest and pythonic way of doing it. If we merge the elements of the list into a string so we have something like '0100001' we can easily check if the combination '0000' is in the string by using the "in" keyword. So that is what we did. We made a function that converted each element of an array into a string and then merged them all together:

```
1.  # Converts and array into a string joining all the elements
2.  def list_stringify(list):
3.    return ''.join([str(ele) for ele in list])
```

Lastly we just had to see if the winning combination is in each of the four directions using all the functions we got till now like this:

```
1.  # Returns if there is a combination of 4 on-line 'player' codes in any of the
2.  # four directions
3.  def check_winner(mat, r, c, player, on_line=4):
4.    winner_sequence = str(player) * on_line
5.    return (
6.      winner_sequence in list_stringify(get_row(mat, r, c)) or
7.      winner_sequence in list_stringify(get_col(mat, r, c)) or
8.      winner_sequence in list_stringify(get_dig(mat, r, c)) or
9.      winner_sequence in list_stringify(get_cdg(mat, r, c))
10.   )
```

Calling this function each time a move is played, giving it the coordinates of where the new chip has been played and the code of the player (either 0 or 1) it will return if a 4-on-line combination is made and with that we can control the game flow. By continuing the game if there is no combination and ending it if there is.

UPDATE:

When we had the program finished we thought it would be a good addition to, if there is a sequence, mark it drawing the borders of the circles that make the sequence with a color. For that we need to return the coordinates of each sequence. We first modified the functions that constructs the list of each direction to return the list of the values of the board in such direction and in addition a list with the coordinates that correspond to each element of the first list, like this:

```
1.  # Returns the values and de coordinates of the row at the position r and c
2.  def get_row(mat, r, c):
3.    row = mat[r]
4.    coords = [(r, i) for i in range(len(row))]
5.    return row, coords
6.
7.  # Returns the values and de coordinates of the column at the position r and c
8.  def get_col(mat, r, c):
9.    col = [row[c] for row in mat]
10.   coords = [(i, c) for i in range(len(mat))]
11.   return col, coords
12.
13. # Returns the values and de coordinates of the diagonal at the position r and c
14. def get_dig(mat, r, c):
15.   off = c - r
16.   coords = [(i, i+off) for i in range(len(mat)) if 0 <= i+off < len(mat[i])]
17.   dig = [mat[row][col] for row, col in coords]
18.   return dig, coords
19.
20. # Returns the values and de coordinates of the counter diagonal at r and c
```

```
def get_cdg(mat, r, c):
  flip_horizontal_mat = [row[::-1] for row in mat]
  width = len(mat[0]) - 1
  new_c = width - c
  cdg, coords = get_dig(flip_horizontal_mat, r, new_c)
  # Flip de column coordinates back
  coords = [(row, width-col) for row, col in coords]
  return cdg, coords
```

Then we wrote a new function to replace the line:

```
winner_sequence in list_stringify(get_row(mat, r, c))
```

Because we need to know the index where the sequence starts and when it ends so we can get the right coordinates from the coords list. For that we can use the method str.index() that returns the index where the sequence given as a parameter appears in the string. If the sequence doesn't appear then it throws a ValueError so we must encapsulate everything in a try except block. After we have the index we just must return the sub list of the cords list starting from the index until the index plus the length of the sequence, in this case is 4:

```
# Returns the list of the coordinates that corresponds to the values that maches
# the sequence. Returns an empty list if the sequence is not present in the
# values
def check_sequence(values, coords, sequence):
  try:
    index = list_stringify(values).index(sequence)
    return coords[index:index + len(sequence)]
  except ValueError:
    return []
```

Then we must modify the check_winner function to call this new check_sequence function like this:

```
# Returns if there is a combination of 4-on-line 'player' codes in any of the
# four directions
def check_winner(mat, r, c, player, on_line=4):
  sequence = str(player) * on_line
  # Check the four directions, if there is a sequence in any of them return a
  # list containing the coordinates of the positions that make the sequence
  row = check_sequence(*get_row(mat, r, c), sequence)
  if row: return row
  col = check_sequence(*get_col(mat, r, c), sequence)
  if col: return col
  dig = check_sequence(*get_dig(mat, r, c), sequence)
  if dig: return dig
  cdg = check_sequence(*get_cdg(mat, r, c), sequence)
  if cdg: return cdg
  # Return an empty list otherwise
  return []
```

4. A.I. Strategy

The most basic strategy would be to play at a random valid column every time. But we can do better than this. In-fact since connect four is a game with perfect information it is a solved game and there exists a strategy that wins every time for the player that makes the first move. But we don't need to implement the perfect strategy either, it will be fine to do something better than just random.

The two most simple rules that comes to mind are, when going to play a chip, check if there is a position where you can win immediately, if there isn't check if there is a position where if isn't blocked

the enemy will win immediately after your move. Finally, if neither of the previous conditions exists play a random move:

```
# A.I. Given a state matrix and the player code calculates the best next move
def computer play(mat, top, player):
  # Try making 4-on-line for each of the columns possible
  for col, row in enumerate(top):
    if row < len(mat):
      mat[row][col] = player
      winner = check_winner(mat, row, col, player)
      mat[row][col] = 2
      if winner:
        return col
  # Switch the player
  player ^= 1
  # See if the enemy could make 4-on-line for each of the columns possible
  for col, row in enumerate(top):
    if row < len(mat):
      mat[row][col] = player
      winner = check_winner(mat, row, col, player)
      mat[row][col] = 2
      if winner:
        return col
  # If doesn't find any of the previous combinations play a random move
  return randint(0, len(mat[0])-1)
```

5. Others

We consider that the rest of the code is very straightforward and self-explanatory and that there is no need to explain it further.