

WASHINGTON STATE UNIVERSITY VANCOUVER

PROGRAM DESIGN AND DEVELOPMENT - CS 121

---

## Assignment 9 (Final Prep)

---

*Professor:*  
Ben MCCAMISH

November 9, 2020

# Overall Assignment

---

Your code should all be contained in a single file titled `assignment9.py`. Your program must be written for Python 3.6+.

**Warning:** This assignment will be used for the final project. You will want to make sure that it works before moving onto the final assignment.

This week you will implement the Connect 4 game. Connect Four is a variation of tic-tac-toe played on a 7x6 rectangular board. The game is played by two players, alternating turns, with each trying to place four checkers in a row vertically, horizontally, or diagonally. One constraint in the game is because the board stands vertically, the checkers cannot be placed in any arbitrary position. A checker may only be placed at the top of one of the currently existing columns (or it may start a new column).

## 1 Getting started

---

Create a new file called `assignment9.py` and create a class called `Connect4` with a constructor method and a representation method. Alternatively, you may use the template provided in autolab.

### 1.1 Constructing the Class (5 points)

Examine the following code (included in the handout). We will need some data structure to store the board. A good one to use here might be a list of lists. That way, we can have a 2d board for the game. Our constructor will accept a width and a height. This is assuming the game could be played on a size other than the default 6x7. In total, the `Connect4` class should have three variables, the board (a list of lists), height, and width.

```
class Connect4(object):
    """This is the Connect4 Constructor"""
    def __init__(self, width, height, window=None):
        self.width = width
        self.height = height
        self.data = []

        for row in range(self.height):
            boardRow = []
            for col in range(self.width):
                boardRow += [' ']
            self.data += [boardRow]
```

### 1.2 Representing the Class

We will want a clean way of presenting the class. Examine the `__repr__` method in the handout. Notice how it is missing some code that you will need to fill in. Once your code is filled it, your object should print like the following:

```
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
-----
0 1 2 3 4 5 6
```

## 2 Performing Actions

---

First you will want to implement some methods that can perform actions. We will start with the most interesting one, adding a move.

### 2.1 Adding Moves (10 points)

Now we will need some method that can add moves to our game. This method takes two inputs: the first input `col` represents the index of the column to which the checker will be added; the second input `ox` either an 'X' or 'O'. In `addMove` you do not have to check that `col` is a legal column number or that there is space in column `col`. That checking is important, however. The next method, which is called `allowsMove`, will do just that. You will want to call this method inside of `addMove` to determine if the move can be performed.

```
def addMove(self, col, ox ):
    #find the first row in the column
    #without a checker in it and
    #then add the ox checker there...
    #do this by checking values
    #in self.data...
```

#### Example

Suppose I have the following code:

```
b = Connect4(7,6)
b.addMove(0,'X')
b.addMove(1,'O')
b.addMove(1,'X')
b.addMove(2,'O')
print(b)
```

I should see this output:

```
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| |X| | | | |
|X|O|O| | | |
-----
0 1 2 3 4 5 6
```

### 2.2 Clear Board (5 points)

The method `clear(self)`, should clear the board that invokes it. Not much to say about `clear(self)`. It may be useful when starting a new game.

### 2.3 Delete Move (20 points)

The method `delMove(self, c)` removes a checker from the board. This method should do the opposite of `addMove`. It should remove the top checker from the column `c`. If the column is empty, then `delMove` should do nothing. This function may not seem useful now, but it will become very useful when you try to implement your own Connect Four player (Final Project).

## 3 Performing Checks

---

In this section you will implement two new methods that perform checks on your game board. The first will check whether the move tried is legal or not. The second will check if the board is full. We will be using these later for prompts with the user.

### 3.1 Allows Move (5 points)

The method `allowsMove(self, c)` is used for checking if a column is a legal move. This method should return `True` if the calling object (of type `Connect4`) does allow a move into column `c`. It returns `False` if column `c` is not a legal column number for the calling object. It also returns `False` if column `c` is full.

### 3.2 Board Full (5 points)

The method `isFull(self)` checks if the board is full. This method should return `True` if the calling object (of type `Connect4`) is completely full of checkers. It should return `False` otherwise. Notice that you can leverage `allowsMove` to make this method very concise! Unless you're supernaturally patient, you'll want to test this on small boards.

## 4 Winning The Game (30 points)

---

Next, you will want to implement another method that can check whether the game has been won or not. The method `winsFor(self, ox)` checks if someone has won the game. It should return `True` if there are four checkers of type `ox` in a row on the board. It should return `False` otherwise. One way to approach this is to consider each possible “anchor” checker that might start a four-in-a-row run. For example, all of the “anchors” that might start a horizontal run (going from left to right) must be in the columns *at least four places from the end of the board*. For example, the following code will check for wins horizontally. You will want to implement the remainder of the function that tests for wins Vertically, NE↔SW, and NW↔SE.

```
def winsFor(self, ox):
    for row in range(self.height):
        for col in range(self.width - 3):
            if self.data[row][col] == ox and \
                self.data[row][col+1] == ox and \
                self.data[row][col+2] == ox and \
                self.data[row][col+3] == ox:
                return True
```

## 5 Hosting the Game (20 points)

---

The method `hostGame(self)` brings everything together into the familiar game. It should alternate turns between ‘X’ and ‘O’. It should ask the user to select a column number for each move. Here are a few important points to keep in mind:

- This method should print the board before prompting for each move.
- After obtaining a move, you should check if the column chosen is a valid one (using `allowsMove()`). If invalid, issue an error message and prompt the user for another move instead.
- This method should place the checker into its (valid) column. Then it should check if that player has won the game or if the board is now full.
- If the game is over for either reason, the game should stop, the board should be printed out one last time, and the program should report who won (or that it was a tie.)

## Autolab Notes:

---

- Submit your python code as you go to determine whether you answered the sections correctly (highly suggested).
- The tests for the `hostgame()` will be performed by the TA.
- Even though points are assigned to certain functions, others must be working to earn them. For example, `clear()` won't work unless `addMove()` is finished, as I cannot clear a board without first adding some things to it.
- Only submit a single .py file.