Today: Connect Four



http://www.coolmath-games.com/0-4-in-a-row

Classes: DIY data (review)

design-it-yourself!



Class: a user-defined datatype

Object: data or a variable whose type is a class

Classes: DIY data (review)



Class: a user-defined datatype

Object: data or a variable whose type is a class

```
d = Date(5, 16, 2021)
d.tomorrow()
print(d)
```

constructor

object

method

uses repr

d would be named **self** inside the **Date** class...

Method: a function defined *in a class* called *by an object*

`self`: in a class, the name of the **object** calling a method

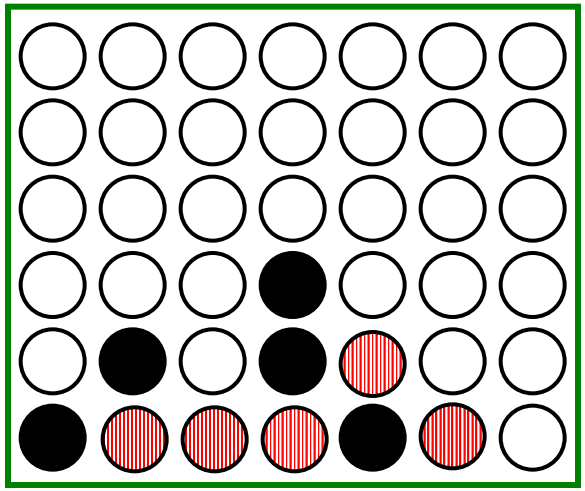
Constructor: the **`__init__`** method for creating a new object

repr: the **`__repr__`** method; returns a string to print

data member: the data in **`self`**: `self.day, self.month, self.year`

Why classes?

Python has no Connect-Four data type…



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

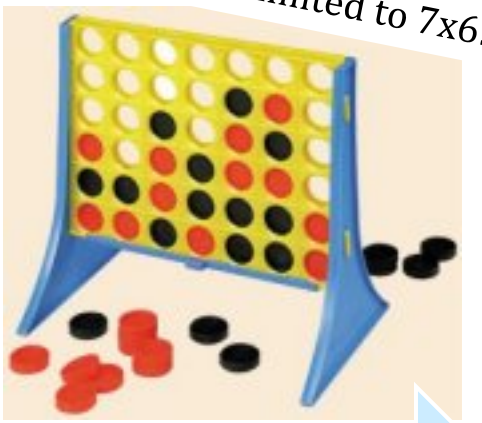


Care for a game?

… but now we can fix that!

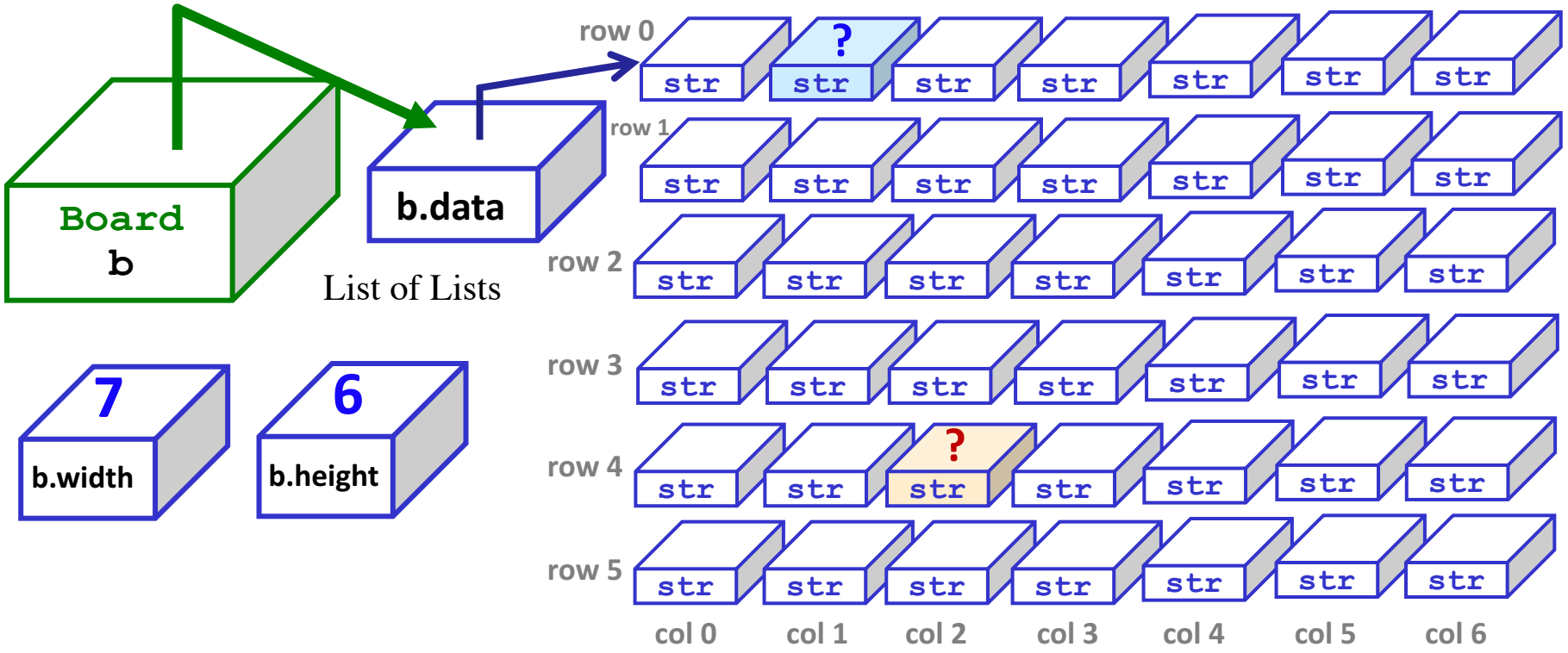
Data design…

(**Data Members**) What data do we need?



(**Methods**) What are the capabilities we want?

Our Board object, `b`

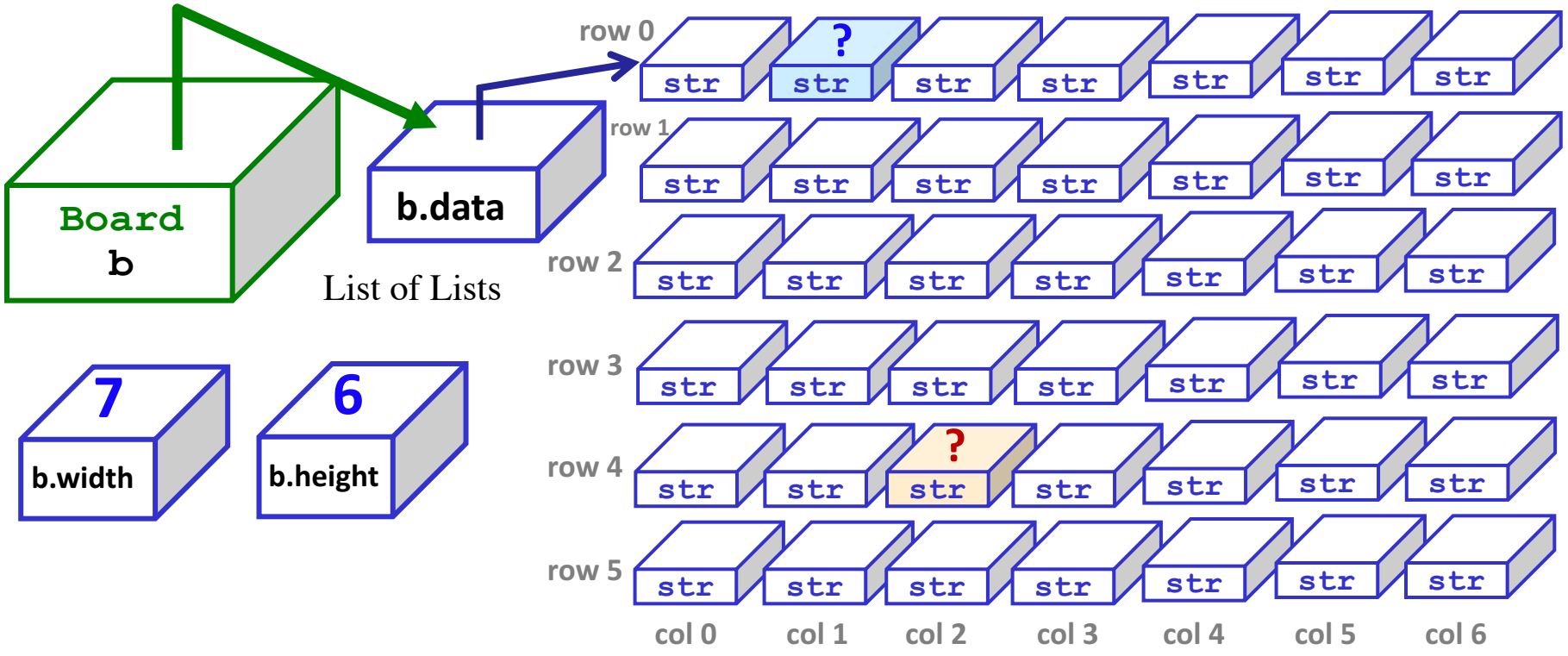


`b.`

`b.`

How could we set ? to **'X'** and ? to **'O'**

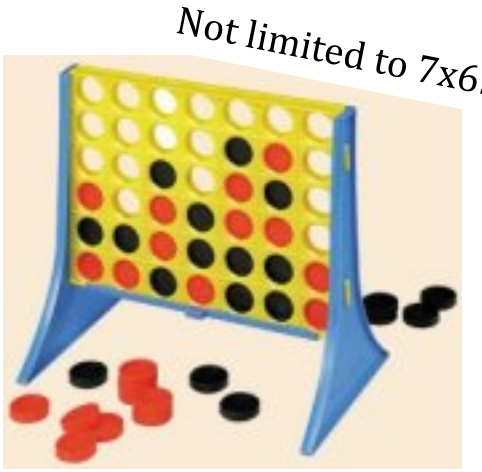
Our Board object, `b`




```
b.data[0][1] = 'X'
```

```
b.data[4][2] = 'O'
```

Data design…



(**Data Members**) What data do we need?

(**Methods**) What are the capabilities we want?

__init__

the "constructor"

```
class Board(object):
    """A data type representing a Connect-4 board
       with an arbitrary number of rows and columns."""

    def __init__(self, width, height):
        """Construct objects of type Board with
           the given width and height."""
        self.width = width
        self.height = height
        self.data = [[' ']*width for row in range(height)]
```

__init__

the "constructor"

```
class Board(object):
    """A data type representing a Connect-4 board
       with an arbitrary number of rows and columns."""

    def __init__(self, width, height):
        """Construct objects of type Board with
           the given width and height."""
        self.width = width
        self.height = height
        self.data = [[' ']*width for row in range(height)]
```


This list comprehension lets us create **height** independent rows with **width** independent columns each.

__init__

the "constructor"

```
class Board(object):
    """A data type representing a Connect-4 board
       with an arbitrary number of rows and columns."""

    def __init__(self, width, height):
        """Construct objects of type Board with
           the given width and height."""
        self.width = width
        self.height = height
        self.data = []
        for row in range(height):
            self.data.append([' ']*width)
```

Can also use a for loop!

list.append()
method on list objects
Adds an element to a list

__repr__

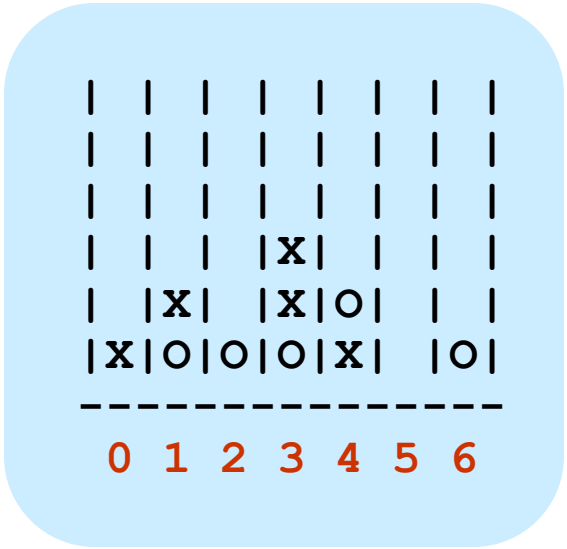
```
def __repr__(self):
    """This method returns a string representation
       for an object of type Board."""

    s = ''
    for r in range(self.height):
        s += '|'
        for c in range(self.width):
            s += self.data[r][c] + '|'
        s += '\n'

    s += (2*self.width + 1) * '-'

    # add the numbers underneath here

    return s
```



__repr__

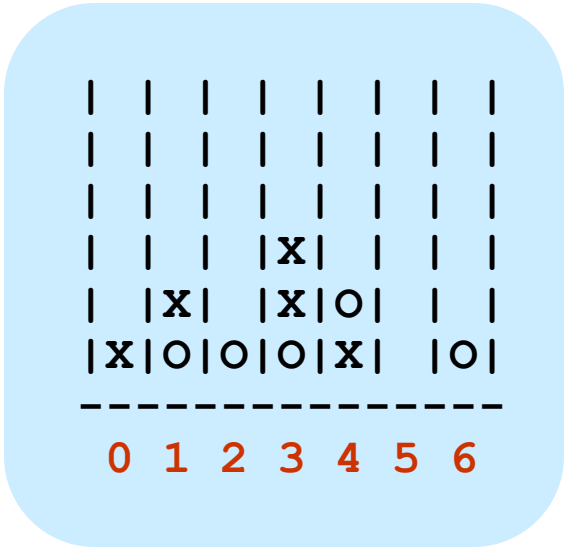
```
def __repr__(self):
    """This method returns a string representation
       for an object of type Board."""

    s = ''
    for r in range(self.height):
        s += '|'
        for c in range(self.width):
            s += self.data[r][c] + '|'
        s += '\n'

    s += (2*self.width + 1) * '-'

    # add the numbers underneath here

    return s
```



Note: __repr__ returns a string.
It does not print anything directly!

addMove

```
class Board(ob
  def addMove(self, col, ox):
      """Drops a checker onto the board."""
      for row in range(0, self.height):
          if self.data[row][col] != ' ':
              self.data[row-1][col] = ox
              return

      self.data[self.height-1][col] = ox
```

a C4 board

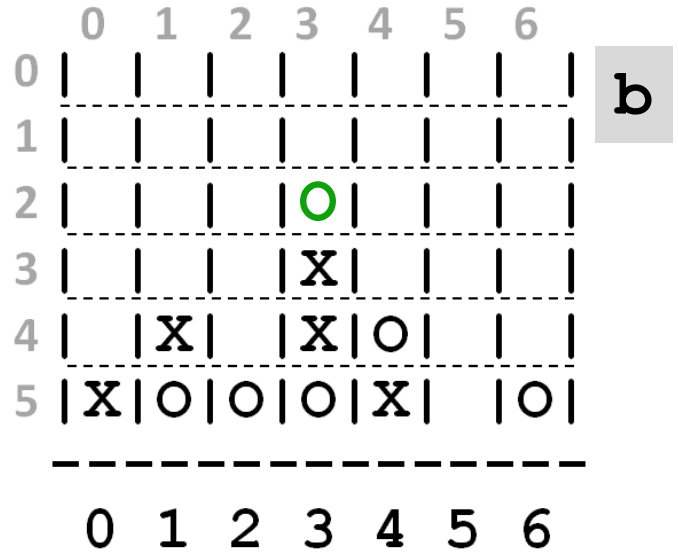
col #

'X' or 'O'

stop the loop!
stop the function!

only gets here if the column is empty—it sets the bottom spot to the correct checker

Run `b.addMove(3, 'O')`



b2.addMoveBUG(3,'O')

addMove

```
class Board(ob
  def addMove(self, col, ox):
      """Drops a checker onto the board."""
      for row in range(0, self.height):
          if self.data[row][col] != ' ':
              self.data[row-1][col] = ox
              return

      self.data[self.height-1][col] = ox
```

a C4 board

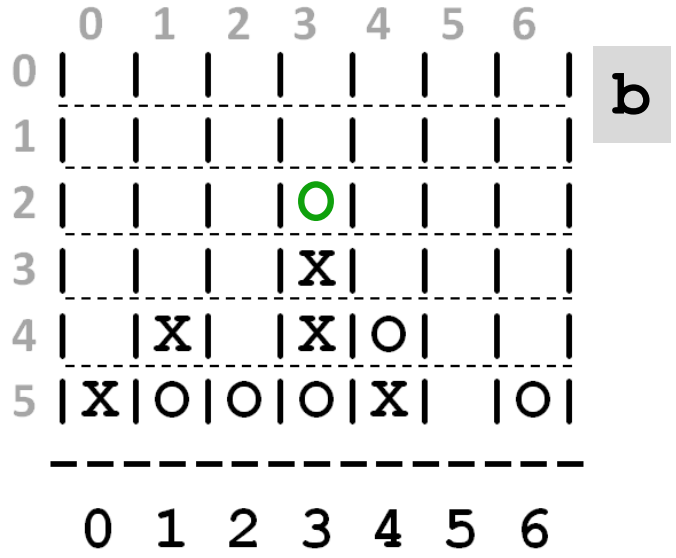
col #

'X' or 'O'

stop the loop!
stop the function!

only gets here if the column is empty—it sets the bottom spot to the correct checker

Run **b.addMove(3, 'O')**



What about error checking?
Illegal column number?
Is there enough space in the column?

We'll handle that in another method: `allowsMove`

b2.addMoveBUG(3,'O')

Incremental Testing

Here is a sequence for testing `addMove`:

```
b = Board(7, 6)
b.addMove(0, 'X')
b.addMove(0, 'O')
b.addMove(0, 'X')
b.addMove(3, 'O')
b.addMove(4, 'O') # Cheat by letting O go again!
b.addMove(5, 'O')
b.addMove(6, 'O')
print(b)
```

Should give you:

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

Let's finish this **allowsMove** method ...

```
class Board(objec
  def allowsMove(self, col):
      """True if col is in-bounds & open
         False otherwise"""
      H = self.height
      W = self.width
      D = self.data

      if
         return False

      elif
         return False
      else:
         return True
```

a C4 board

col #

try these shortcuts

out of bounds?

col full?

Allowed!

If col is ***out-of-bounds*** or ***full***, return **False**.

If it's *in-bounds and not full*, return **True**.

b

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

```
b.allowsMove(0) == True
b.allowsMove(1) == True
b.allowsMove(2) == False
b.allowsMove(3) == False
b.allowsMove(4) == True
b.allowsMove(5) == True
b.allowsMove(6) == False
b.allowsMove(7) == False
```

Let's finish this **allowsMove** method ...

```
class Board(objec
    def allowsMove(self, col):
        """True if col is in-bounds & open
           False otherwise"""
        H = self.height
        W = self.width
        D = self.data

        if col >= W or col < 0:
            return False

        elif D[0][col] != " ":
            return False
        else:
            return True
```

a C4 board

col #

try these shortcuts

out of bounds?

col full?

Allowed!

If col is ***out-of-bounds*** or ***full***, return **False**.

If it's *in-bounds and not full*, return **True**.

b

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

```
b.allowsMove(0) == True
b.allowsMove(1) == True
b.allowsMove(2) == False
b.allowsMove(3) == False
b.allowsMove(4) == True
b.allowsMove(5) == True
b.allowsMove(6) == False
b.allowsMove(7) == False
```

hw11pr2: **Board** class

✓ the "constructor"	`__init__(self, width, height)`
✓ checks if allowed	`allowsMove(self, col)`
✓ places a checker	`addMove(self, col, ox)`
removes a checker	`delMove(self, col)` ← to write...
✓ outputs a string	`__repr__(self)`
checks if any space is left	`isFull(self)` ← to write...
checks if a player has won	`winsFor(self, ox)` ← to write...
the game...	`hostGame(self)` ← to write...

Which are similar to others? Which requires the most thought?

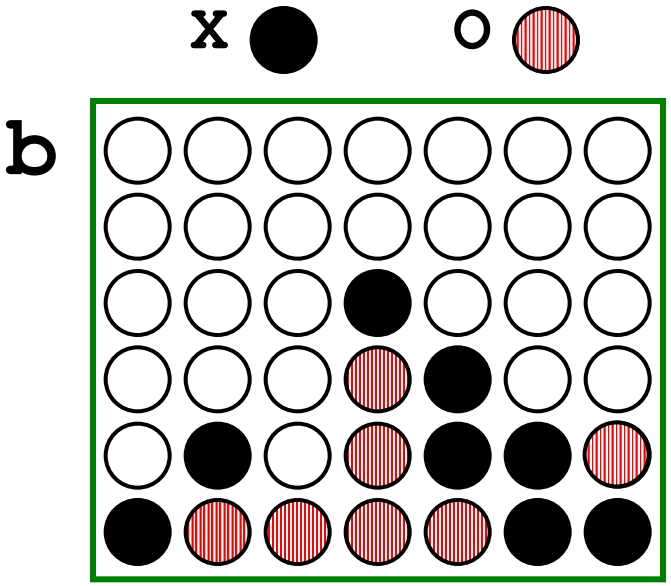
Board class

✓ the "constructor"	`__init__(self, width, height)`
✓ checks if allowed	`allowsMove(self, col)`
✓ places a checker	`addMove(self, col, ox)`
removes a checker	`delMove(self, col)` ← to write...
✓ outputs a string	`__repr__(self)`
checks if any space is left	`isFull(self)` ← to write...
checks if a player has won	`winsFor(self, ox)` ← to write...
the game...	`hostGame(self)` ← to write...

Trickiest ones are `winsFor` and `hostGame`

Which are similar to others? Which requires the most thought?

winsFor(self, ox)




```
b.winsFor('X')
```

or `'O'`

```
def winsFor(self, ox):
    """Does ox win?"""

    for row in range(
      for col in range(
```

See the writeup for hints on implementing this method.



Watch out for *corner cases*!

Why objects and classes?

Elegance: Objects *hide complexity!*

```
if b.winsFor('X') == True:
```

```
if d.isBefore(d2):
```

Simple – and *INVITING* --
building blocks!

