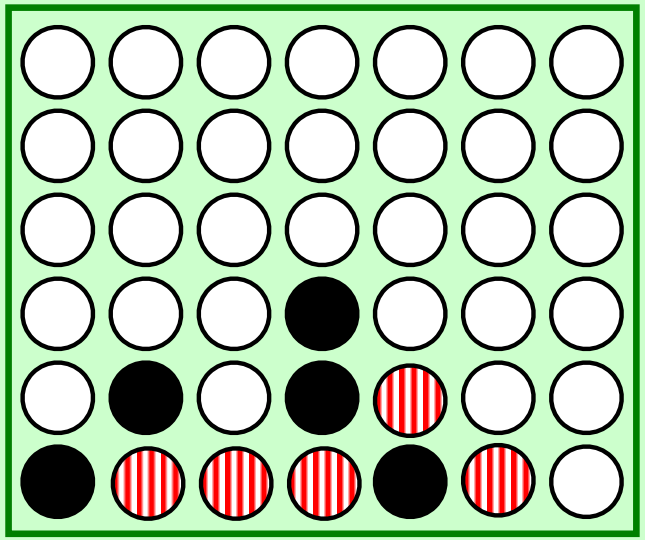
CS 4 this week

Hw #10.2 due 4/**23**

Building classes...

`hw10pr2`

Connect Four **Board** class



If you're board, get to a class!

Classes: DIY data

design-it-yourself!

Class: a user-defined datatype

Object: data or a variable whose type is a class

OOP!

object-oriented programming

Classes: DIY data

design-it-yourself!

Class: a user-defined datatype

Object: data or a variable whose type is a class

constructor

object

```
d = Date( 12, 31, 2018 )
d.tomorrow()
print(d)
```

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

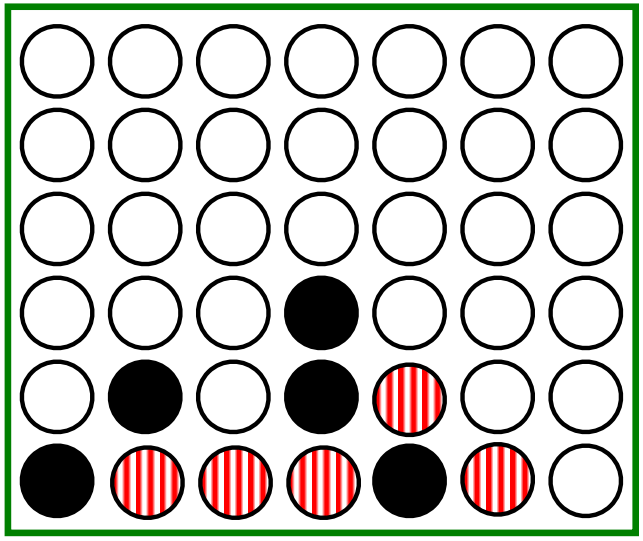
Constructor: the **`__init__`** function for creating a new object

repr: the **`__repr__`** function returning a string to print

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

Why classes?

Python has no Connect-four datatype…



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

b2

Care for a game?

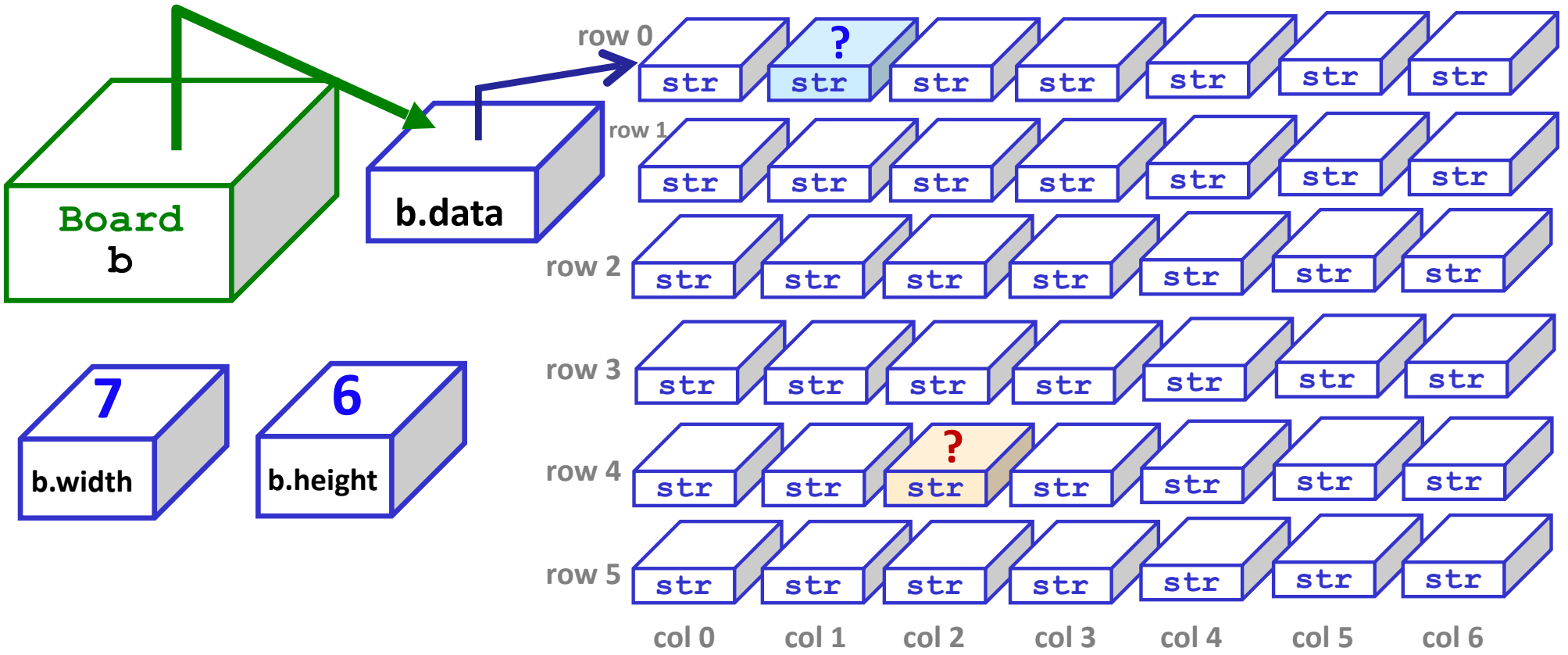
… and now we can fix that!

Data design…

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

(**Methods**) What capabilities do we want?

Our Board object, `b`



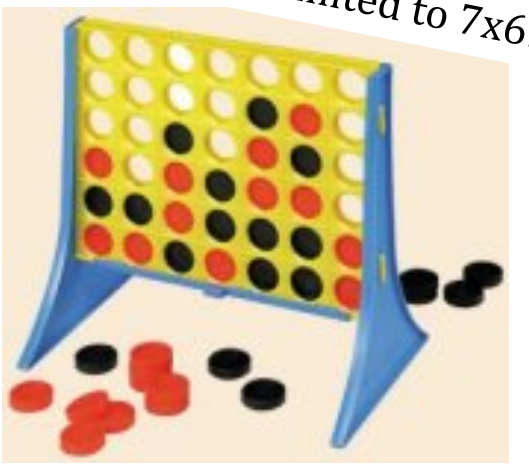
`b.`

`b.`

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

Data design…

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



(**Methods**) What capabilities do we want?

__init__

the "constructor"

```
class Board:
    """ a datatype representing a C4 board
        with an arbitrary number of rows and cols
    """

    def __init__( self, width, height ):
        """ the constructor for objects of type Board """
        self.width = width
        self.height = height
        W = self.width
        H = self.height
        self.data = [ [' ']*W for row in range(H) ]
```

__init__

the "constructor"

```
class Board:
    """ a datatype representing a C4 board
        with an arbitrary number of rows and cols
    """

    def __init__( self, width, height ):
        """ the constructor for objects of type Board """
        self.width = width
        self.height = height
        W = self.width
        H = self.height
        self.data = [ [' ']*W for row in range(H) ]
```

convenient!

This list comprehension lets us create
H independent rows with
W independent columns each.

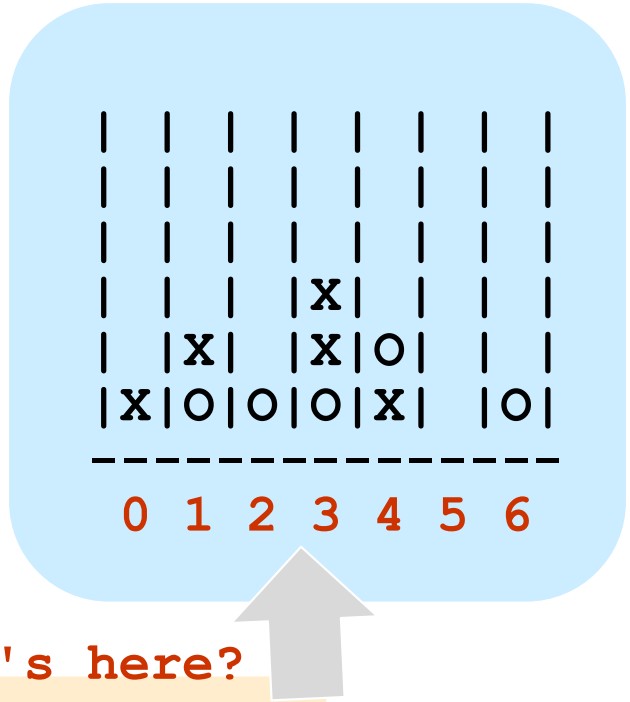
__repr__

```
def __repr__(self):
    """ this method returns a string representation
        for an object of type Board
    """
    H = self.height
    W = self.width
    s = ''
    for r in range( H ):
        s += '|'
        for c in range( W ):
            s += self.data[r][c] + '|'
        s += '\n'

    s += (2*W+1)*'-'

    # what kind of loop will add the col #'s here?

    return s
```



Quiz

```
class Board:

    def addMove(self, col, ox):
        """ buggy version! """
        H = self.height
        for row in range(0,H):
            if self.data[row][col] != ' ':
                self.data[row-1][col] = ox
```

a C4 board

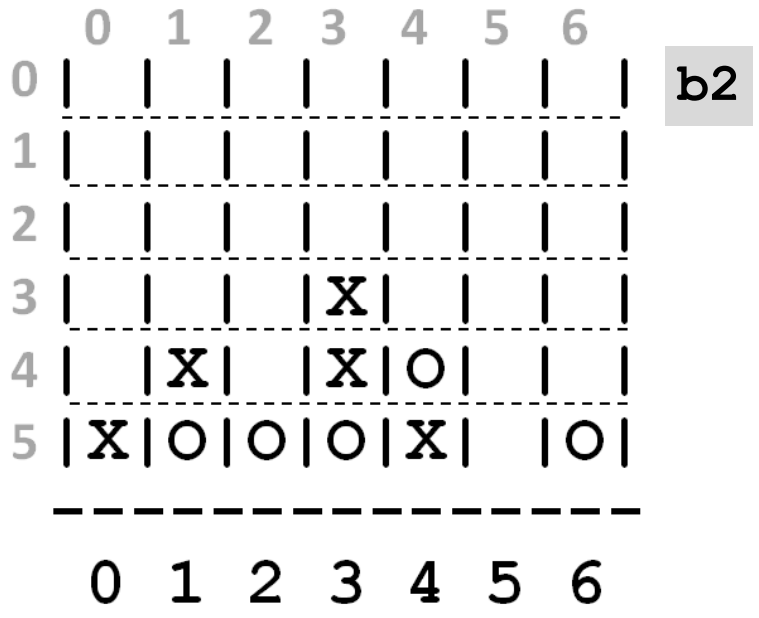
col #

'X' or 'O'

shortcut

(1) Run **`b2.addMove(3,'O')`**

(2) ***Bugs!*** Can you fix them?!



Name(s) _____

Quiz

```
class Board:

    def addMove(self, col, ox):
        """ correct version! """
        H = self.height
        for row in range(0,H):
            if self.data[row][col] != ' ':
                self.data[row-1][col] = ox
                return

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

a C4 board

col #

'X' or 'O'

shortcut

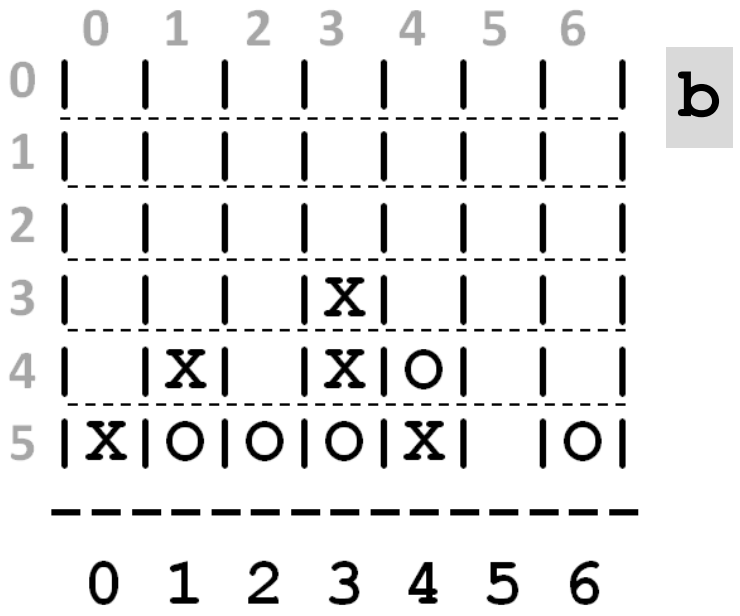
stop the loop! stop the function!

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

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

(2) **Bu** ...em?!

this page has the correct version...

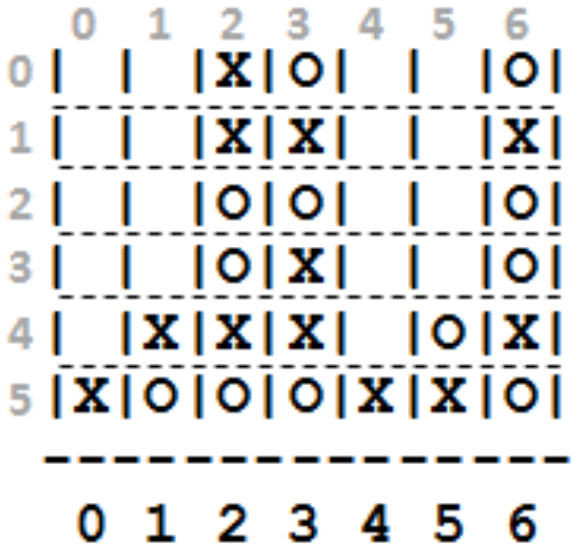


Try this on the back page first...

b2.addMoveBUG(3,'O')

Let's <u>understand</u> this **allowsMove** method ...

b3.allowsMove



b3

```
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
```

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

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

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

```
class Board:

    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 #

shortcuts!

out of bounds?

col full?

Allowed!

b3.allowsMove

b3

```
  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
```

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

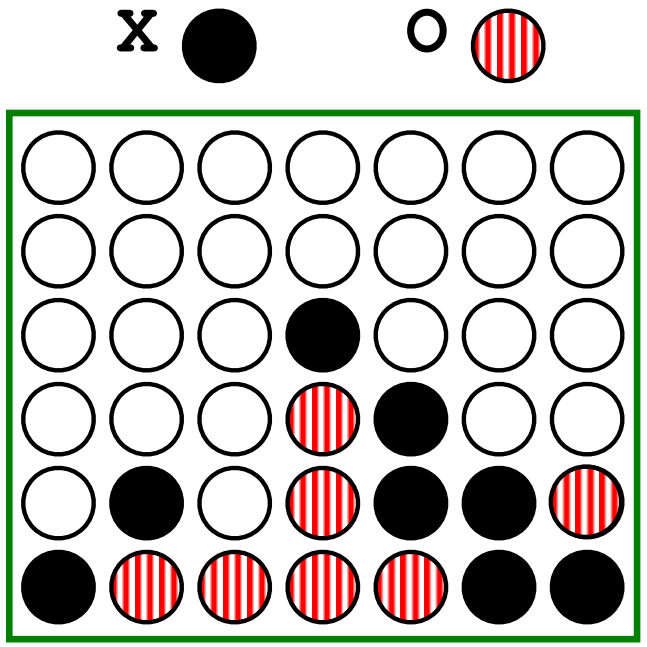
If it's *in-bounds and not full*, return **True**.

hw10pr2: **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?

winsFor(self, ox)




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

or 'O'

```
def winsFor(self, ox):
    """ does ox win? """
    H = self.height
    W = self.width
    D = self.data

    for row in range(
      for col in range(

          if
          if
          if
          if
```

Does this look familiar!?

```
>>> b4.winsFor( 'X' )
True
>>> b4.winsFor( 0 )
False
>>> b4.winsFor( '0' )
False
>>> b4.winsFor( 'O' )
True
```

Watch out for
corner cases!

Why objects and classes?

Elegance: Objects *hide complexity!*

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

```
rem = self.diff( d2 ) % 7
```

Simple – and *INVITING* --
building blocks!