Lecture 12
Inheritance

UTA Irene inherited her good looks from her mom

based in part on notes from the CS-for-All curriculum developed at Harvey Mudd College

Last Time (lectures 10 & 11)

Lecture 10: Object Oriented Programming (OOP)

- Built-in Types (strings, files, etc.)
  - Everything in Python is an object!
- `type()` function
- Objects combine Attributes and Methods
- Classes vs. Instances
- Creating our own objects: `Rectangle`
- The Constructor: `__init__()`
- Referencing internal attributes: `self.x`
- Mutable attributes: `Rectangle.width`
- Methods: `grow()`, `area()`

Project 2: Modeling overview

Last Time (lectures 10 & 11)

Lecture 11: Object Oriented Programming (OOP)

- Importing a custom class from file (e.g., rectangle.py)
  - `from rectangle import *`
- Testing for equality: `a == b`:
  - Default: `id(a) == id(b)`
  - Operator overloading: `a.__eq__(self,b)`
- String representation of object: `__repr__`
  - `>>> a`
  - `str(a)`
  - `print(a)`
- The 4 pillars of OOP
  - Encapsulation
  - Abstraction
  - **Inheritance <= today's lecture!**
  - Polymorphism

Lecture 12 Goals

Quiz 1

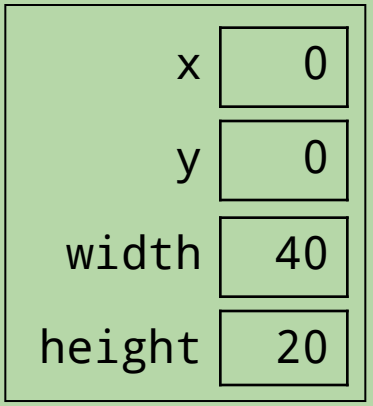
- Overview of class results
- Review of common errors

Inheritance: Special types of objects

Connect 4!

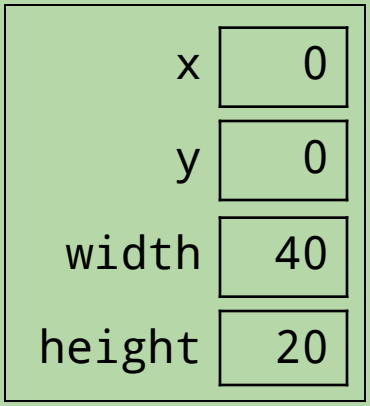
Recall: Rectangle Class

```
class Rectangle:
    def __init__(self, init_width, init_height):
        ...
    def grow(self, dwidth, dheight):
        ...
    def area(self):
         ...
    def perimeter(self):
         ...
    def scale(self, factor):
         ...
    def __eq__(self, other):
         ...
    def __repr__(self):
         ...
```



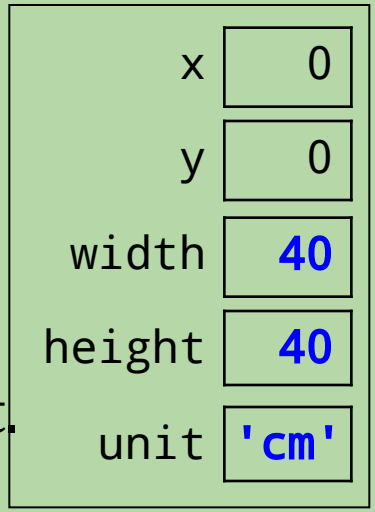
Recall: Using `Rectangle` Methods

```
>>> myrect = Rectangle(40, 20)
>>> print(myrect)
40 x 20
>>> myrect.area()
800
>>> myrect.scale(3)
>>> print(myrect)
120 x 60
```



Squares == Special Rectangles!

- A square has the same basic attributes as a rectangle, but...
  - width and height must be the same
- Assume that we also want `Square` objects to have an attribute for the unit of measurement.



- `Square` objects should behave like `Rectangle`s:

```
>>> mysquare.area()
1600

>>> mysquare.scale(3)
```

- But there should be some differences as well:

```
>>> mysquare = Square(40, 'cm')

>>> print(mysquare)
square with 40-cm sides
```

Using Inheritance

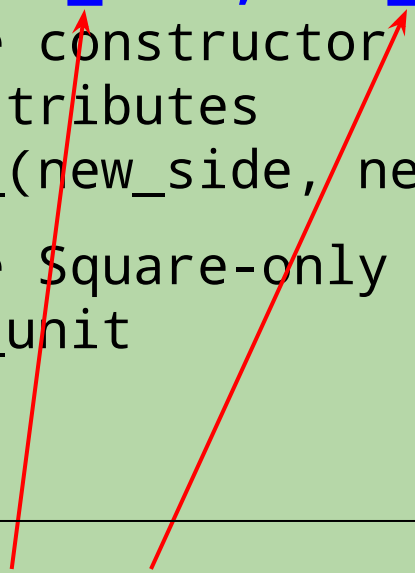
```
class Square(Rectangle):
    def __init__(self, new_side, new_unit):
        # call Rectangle constructor to initialize
        # most of the attributes
        super().__init__(new_side, new_side)

        # initialize the Square-only attribute
        self.unit = new_unit
```

← Square inherits from Rectangle

- `Square` gets all of the attributes and methods of `Rectangle`.
  - we don't need to redefine them here!
- `Square` is a *subclass* of `Rectangle`.
- `Rectangle` is a *superclass* of `Square`.

Constructors and Inheritance

```
class Square(Rectangle):
    def __init__(self, new_side, new_unit):
        # call Rectangle constructor to initialize
        # most of the attributes
        super().__init__(new_side, new_side)

        # initialize the Square-only attribute
        self.unit = new_unit

    ...
```



```
>>> mysquare = Square(40, 'cm')
```

Constructors and Inheritance

```
class Square(Rectangle):
    def __init__(self, new_side, new_unit):
        # call Rectangle constructor to initialize
        # most of the attributes
        super().__init__(new_side, new_side)

        # initialize the Square-only attribute
        self.unit = new_unit

>>> mysquare = Square(40, 'cm')
```

- `super()` provides access to the superclass of the current class.
  - allows us to call its version of `__init__`, which initializes the inherited attributes

Constructors and Inheritance

```
class Square(Rectangle):
    def __init__(self, new_side, new_unit):
        # call Rectangle constructor to initialize
        # most of the attributes
        super().__init__(new_side, new_side)

        # initialize the Square-only attribute
        self.unit = new_unit

>>> mysquare = Square(40, 'cm')
```

- `super()` provides access to the superclass of the current class.
  - allows us to call its version of `__init__`, which initializes the inherited attributes

Using the Inherited Methods

```
class Square(Rectangle):
    def __init__(self, new_side, new_unit):
        # call Rectangle constructor to initialize
        # most of the attributes
        super().__init__(new_side, new_side)

        # initialize the Square-only attribute
        self.unit = new_unit
```

```
>>> mysquare = Square(40, 'cm')
>>> mysquare.area()   # area() is inherited!
1600
>>> print(mysquare)   # __repr__() is inherited!
40 x 40
```

Overriding an Inherited Method

```
class Square(Rectangle):
    def __init__(self, new_side, new_unit):
        super().__init__(new_side, new_side)
        self.unit = new_unit

    def __repr__(self):
        s = 'square with '
        s += str(self.width) + '-' + self.unit
        s += ' sides'
        return s
```

← Square inherits from Rectangle

- To see something different when we print a `Square` object, we *override* (i.e., replace) the inherited version of `__repr__`.

```
>>> mysquare = Square(40, 'cm')
>>> print(mysquare)
???
```

Overriding an Inherited Method

```
class Square(Rectangle):
    def __init__(self, new_side, new_unit):
        super().__init__(new_side, new_side)
        self.unit = new_unit

    def __repr__(self):
        s = 'square with '
        s += str(self.width) + '-' + self.unit
        s += ' sides'
        return s
```

← Square inherits from Rectangle

- To see something different when we print a `Square` object, we *override* (i.e., replace) the inherited version of `__repr__`.

```
>>> mysquare = Square(40, 'cm')
>>> print(mysquare)
square with 40-cm sides
```

Recall: `grow` Method in `Rectangle`

```
class Rectangle:
      ...

    def grow(self, dwidth, dheight):
        self.width += dwidth
        self.height += dheight
```

- This method is inherited by `Square` objects.

- It could be used in problematic ways!

```
>>> sq = Square(40, 'cm')
>>> sq.grow(10, 5)    # shouldn't be allowed!
>>> print(sq.width, sq.height)
50 45
```

Overriding the Inherited `grow()` Method

```
class Square(Rectangle):
    ...
    def grow(self, dwidth, dheight):
        if dwidth != dheight:
            print('invalid change for a square')
        else:
            super().grow(dwidth, dheight)
```

- The new `grow()` has the same arguments as the inherited one.
- It prints an error message if the requested change is invalid:

```
>>> sq = Square(40, 'cm')
>>> sq.grow(10, 5)
invalid change for a square
```

Overriding the Inherited `grow()` Method

```
class Square(Rectangle):
    ...
    def grow(self, dwidth, dheight):
        if dwidth != dheight:
            print('invalid change for a square')
        else:
            super().grow(dwidth, dheight)
```

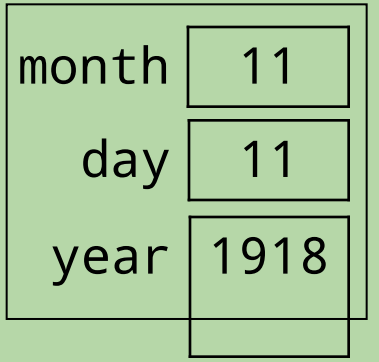
- If the requested change is valid, the new `grow()` calls the version of `grow()` from the superclass to do the work!

Inheritance Continued;

based in part on notes from the CS-for-All curriculum developed at Harvey Mudd College

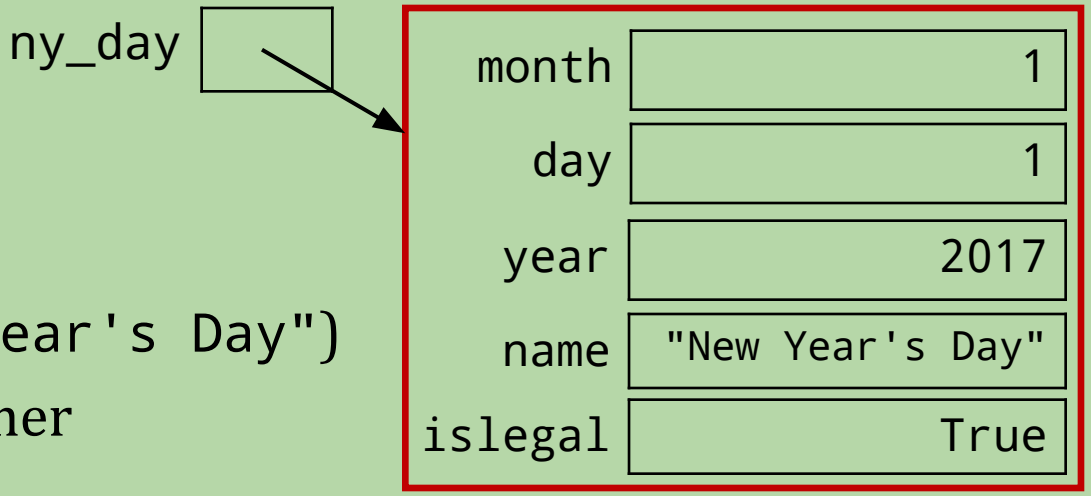
Recall: Our Date Class

```
class Date:
    def __init__(self, new_month, new_day, new_year):
        """ Constructor """
        self.month = new_month
        self.day = new_day
        self.year = new_year

    def __repr__(self):
        """ This method returns a string representation for the
            object of type Date that calls it (named self).
        """
        s =  "%02d/%02d/%04d" % (self.month, self.day, self.year)
        return s

    def is_leap_year(self):
        """ Returns True if the calling object is
            in a leap year. Otherwise, returns False.
        """
        if self.year % 400 == 0:
            return True
        elif self.year % 100 == 0:
            return False
        elif self.year % 4 == 0:
            return True
        return False
```



Holidays == Special Dates!

- Each holiday has:
  - a month
  - a day
  - a year
  - a name (e.g., `"New Year's Day"`)
  - an indicator of whether it's a legal holiday



- We want `Holiday` objects to have `Date`-like functionality:

```
>>> ny_day = Holiday(1, 1, 2017, "New Year's Day")
>>> today = Date(11, 28, 2016)
>>> ny_day.diff(today)
34
```

- But we want them to behave differently in at least one way:

```
>>> print(ny_day)
New Year's Day (01/01/2017)
```

Which statement uses the correct terminology?

```
class Holiday(Date):
    def __init__(self, month, day, year, name):
        ...
```

← Holiday inherits from Date

A. `Date` is a superclass of `Holiday`.

B. `Date` is a subclass of `Holiday`.

C. `Date` is an uberclass of `Holiday`.

D. none of the above

Which statement uses the correct terminology?

```
class Holiday(Date):
    def __init__(self, month, day, year, name):
        ...
```

← Holiday inherits from Date

A. **Date is a superclass of Holiday.**

B. Date is a subclass of Holiday.

C. Date is an uberclass of Holiday.

D. none of the above

Holiday is a subclass of Date.

Which statement uses the correct terminology?

```
class Holiday(Date):
    def __init__(self, month, day, year, name):
        ...
```

← Holiday inherits from Date

A. **`Date` is a superclass of `Holiday`.**

B. `Date` is a subclass of `Holiday`.

C. `Date` is an uberclass of `Holiday`.

D. none of the above

uberclass is not actually a thing...

Let Holiday Inherit From Date!

```
class Holiday(Date):
    def __init__(self, month, day, year, name):
        # call Date constructor to initialize month,day,year
        super().__init__(month, day, year)

        # initialize the non-inherited fields
        self.name = name
        self.islegal = True   # default value

    def __repr__(self):   # overrides the inherited __repr__
        s = self.name
        mdy = ____________________  # use inherited __repr__
        s += ' (' + mdy + ')'
        return s

>>>  ny_day = Holiday(1, 1, 2017, "New Year's Day")
```

← Holiday inherits from Date

Let Holiday Inherit From Date!

```
class Holiday(Date):
    def __init__(self, month, day, year, name):
        # call Date constructor to initialize month,day,year
        super().__init__(month, day, year)

        # initialize the non-inherited fields
        self.name = name
        self.islegal = True   # default value

    def __repr__(self):   # overrides the inherited __repr__
        s = self.name
        mdy = ____________________  # use inherited __repr__
        s += ' (' + mdy + ')'
        return s
```

← Holiday inherits from Date

Let Holiday Inherit From Date!

```
class Holiday(Date):
    def __init__(self, month, day, year, name):
        # call Date constructor to initialize month,day,year
        super().__init__(month, day, year)

        # initialize the non-inherited fields
        self.name = name
        self.islegal = True   # default value

    def __repr__(self):   # overrides the inherited __repr__
        s = self.name
        mdy = ____________________  # use inherited __repr__
        s += ' (' + mdy + ')'
        return s
```

← Holiday inherits from Date

Let Holiday Inherit From Date!

```
class Holiday(Date):          ← Holiday inherits from Date
    def __init__(self, month, day, year, name):
        # call Date constructor to initialize month,day,year
        super().__init__(month, day, year)

        # initialize the non-inherited fields
        self.name = name
        self.islegal = True   # default value

    def __repr__(self):   # overrides the inherited __repr__
        s = self.name
        mdy = ____________________  # use inherited __repr__
        s += ' (' + mdy + ')'
        return s
>>> print(ny_day)
New Year's Day (01/01/2017)
```

How can we call the inherited __repr__?

```
class Holiday(Date):
    def __init__(self, month, day, year, name):
        # call Date constructor to initialize month,day,year
        super().__init__(month, day, year)

        # initialize the non-inherited fields
        self.name = name
        self.islegal = True   # default value

    def __repr__(self):   # overrides the inherited __repr__
        s = self.name
        mdy = ____________________  # use inherited __repr__
        s += ' (' + mdy + ')'
        return s
```

← Holiday inherits from Date

A. `Date.__repr__()`

B. `super().__repr__()`

C. `super().repr(self)`

D. none of the above

How can we call the inherited __repr__?

```
class Holiday(Date):
    def __init__(self, month, day, year, name):
        # call Date constructor to initialize month,day,year
        super().__init__(month, day, year)

        # initialize the non-inherited fields
        self.name = name
        self.islegal = True   # default value

    def __repr__(self):   # overrides the inherited __repr__
        s = self.name
        mdy = super().__repr__()  # use inherited __repr__
        s += ' (' + mdy + ')'
        return s
```

← Holiday inherits from Date

A. `Date.__repr__()`

B. **`super().__repr__()`**

C. `super().repr(self)`

D. none of the above

Let `Holiday` Inherit From `Date`!

```
class Holiday(Date):
    def __init__(self, month, day, year, name):
        # call Date constructor to initialize month,day,year
        super().__init__(month, day, year)

        # initialize the non-inherited fields
        self.name = name
        self.islegal = True   # default value

    def __repr__(self):   # overrides the inherited __repr__
        s = self.name
        mdy = super().__repr__()  # use inherited __repr__
        s += ' (' + mdy + ')'
        return s
```

← Holiday inherits from Date

- That's it! Everything else is inherited!
- All other `Date` methods work the same on `Holiday` objects as they do on `Date` objects!

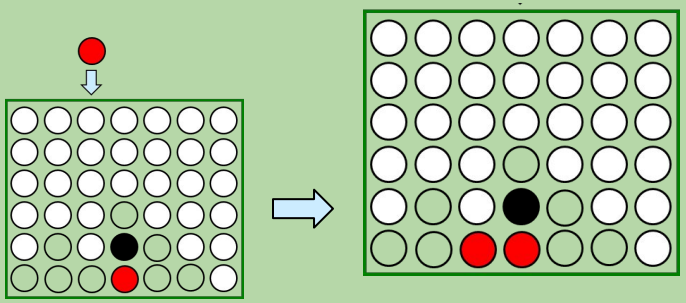
Board and Player Objects for Connect Four

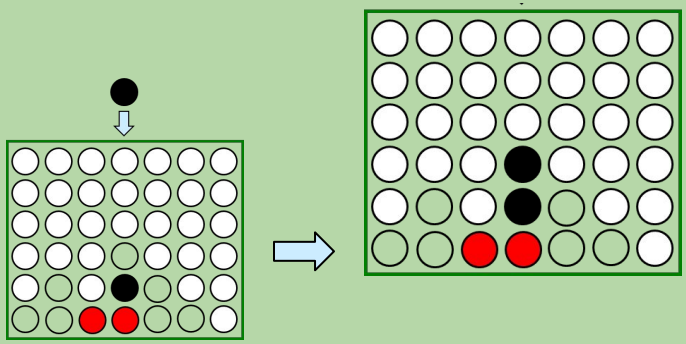
HTA Griffin, age 2, winning at connect four

based in part on notes from the CS-for-All curriculum developed at Harvey Mudd College

hw07: Connect Four!

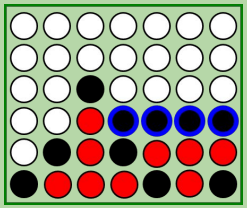
- Two players, each with one type of checker
- 6 x 7 board that stands vertically
- Players take turns dropping a checker into one of the board's columns.

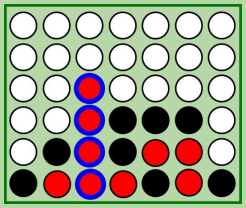


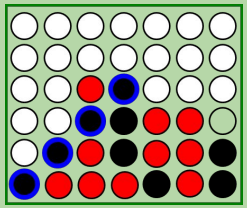


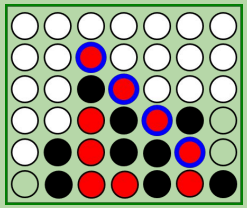
- Win == four adjacent checkers in any direction:

horizontal vertical up diagonal down diagonal









Recall: Classes and Objects

- A *class* is a blueprint – a definition of a new data type.
- We can use the class to create one or more *objects.*
  - "values" / *instances* of that type
- What functions and types of objects could be useful for playing Connect Four with the computer?

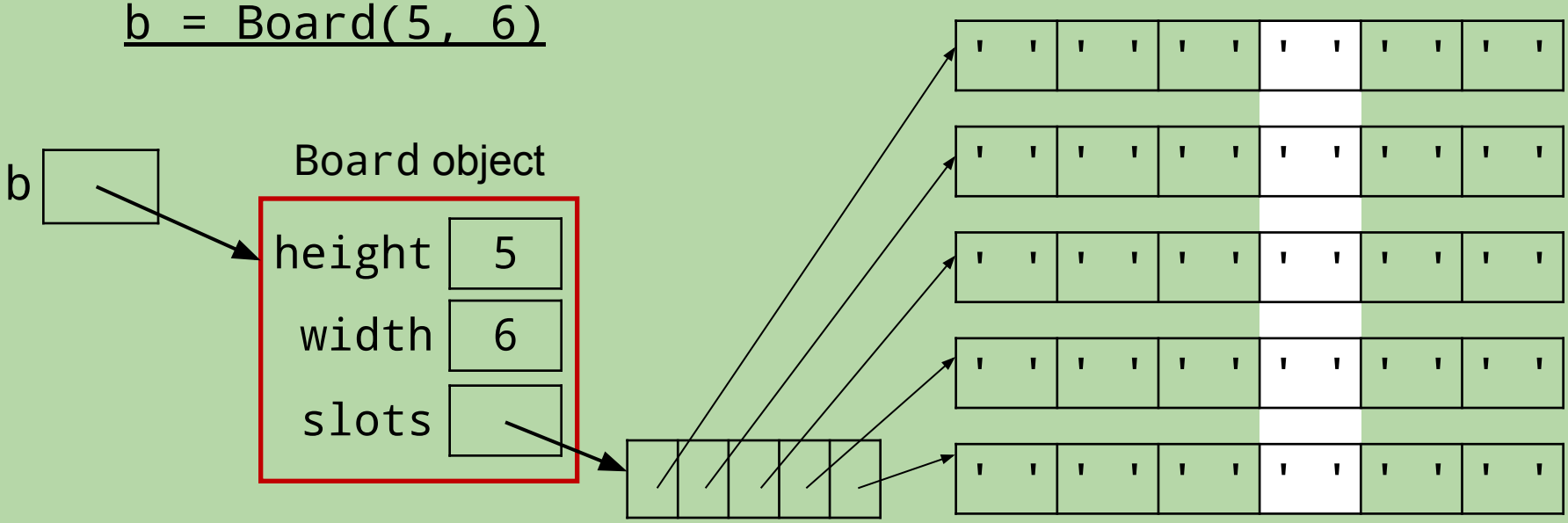
Connect Four Methods

```
def process_move(player, board):
    '''Applies a player objects next move to a board object.
       Returns true if the player wins or a tie occurs,
       False otherwise'''
       pass

def connect_four(player1, player2)
    '''Plays a connect four game between player1 and player2,
       Returns the final board configuration.'''

     while True:  % Play until a win or tie occurs.
        if process_move(player1, board):
            return board

        if process_move(player2, board):
            return board
```

Board Objects

- To facilitate testing, we'll allow for dimensions other than 6 x 7.
  - example: a 5 x 6 board

```
b = Board(5, 6)
```



a 2-D list of single-character strings

`' '` (space) for empty slot

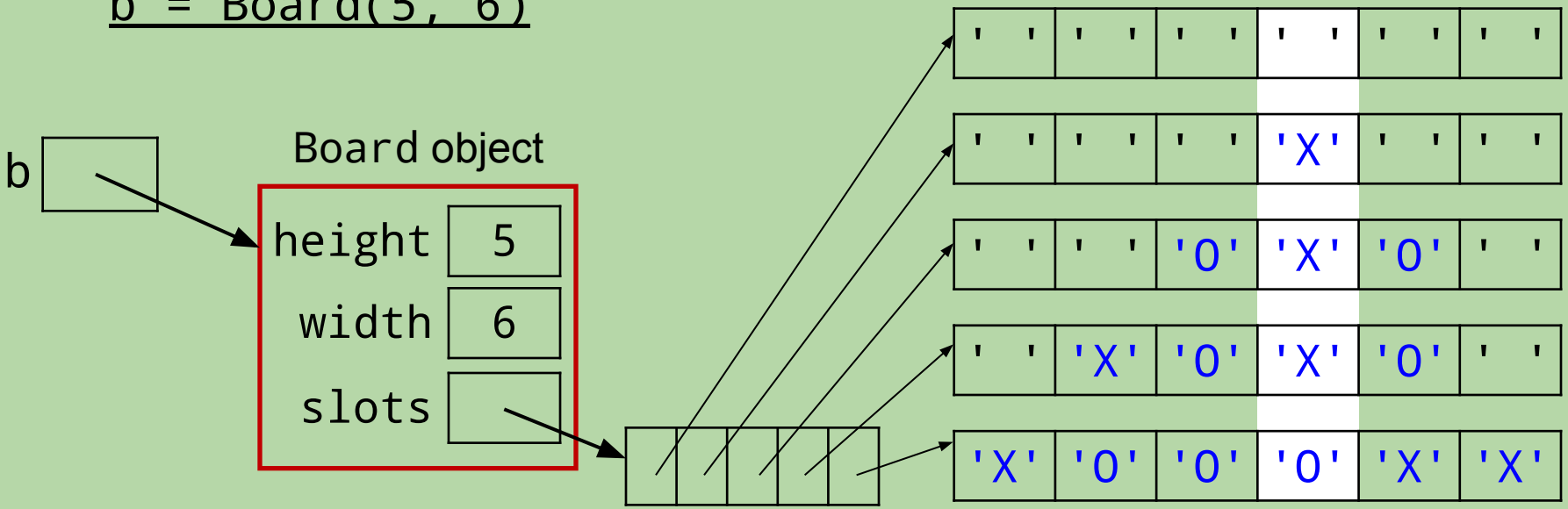
`'X'` for one player's checkers

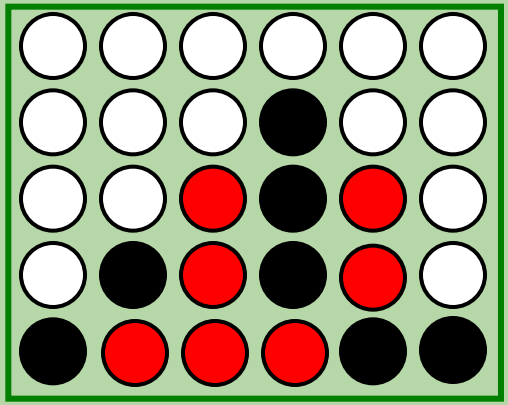
`'O'` (not zero!) for the other's

Board Objects (cont.)

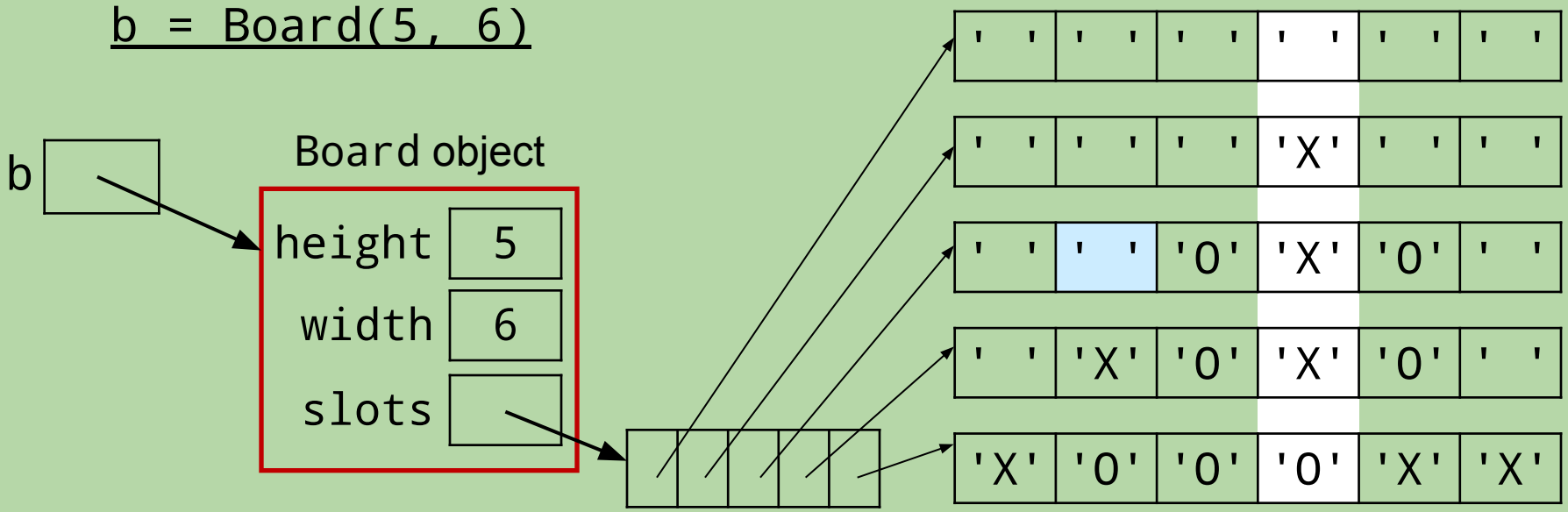
- Here's the same Board after a number of moves:

```
b = Board(5, 6)
```





From a Client, How Could We Set the Blue Slot to 'X'?



A. `Board.slots[3][2] = 'X'`

B. `Board.slots[1][2] = 'X'`

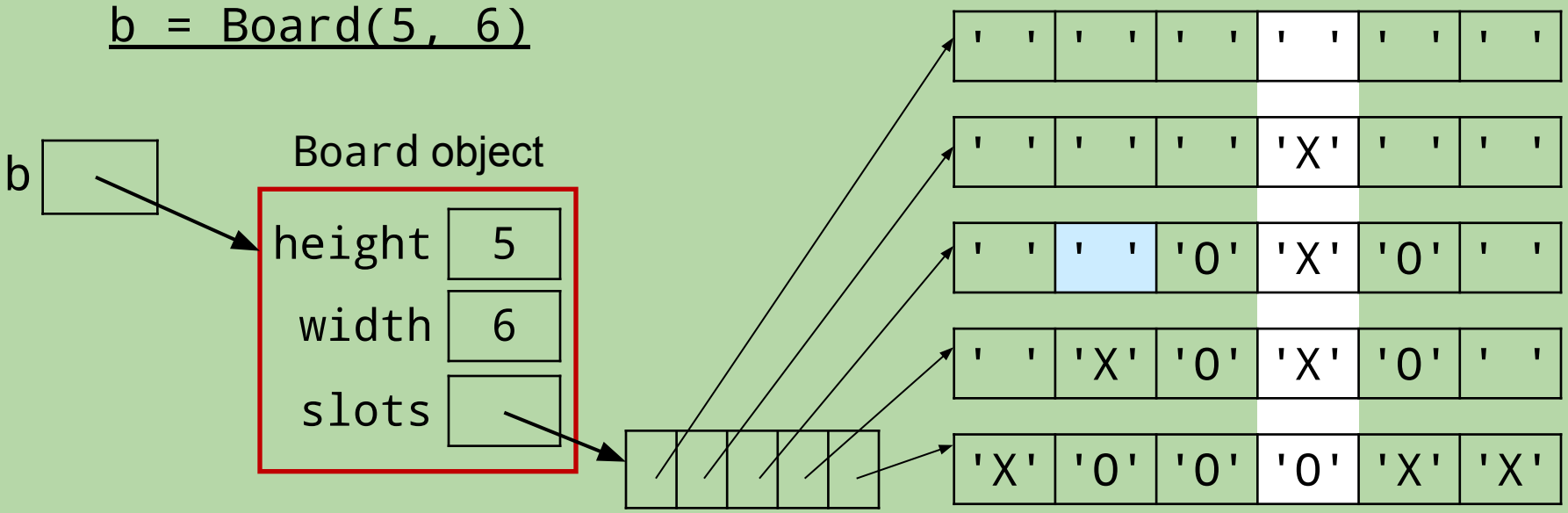
C. `slots[2][1] = 'X'`

D. `b.slots[1][2] = 'X'`

E. `b.slots[2][1] = 'X'`

How would you write this assignment if it were inside a `Board` method?

From a Client, How Could We Set the Blue Slot to 'X'?



A. `Board.slots[3][2] = 'X'`

B. `Board.slots[1][2] = 'X'`

C. `slots[2][1] = 'X'`

D. `b.slots[1][2] = 'X'`

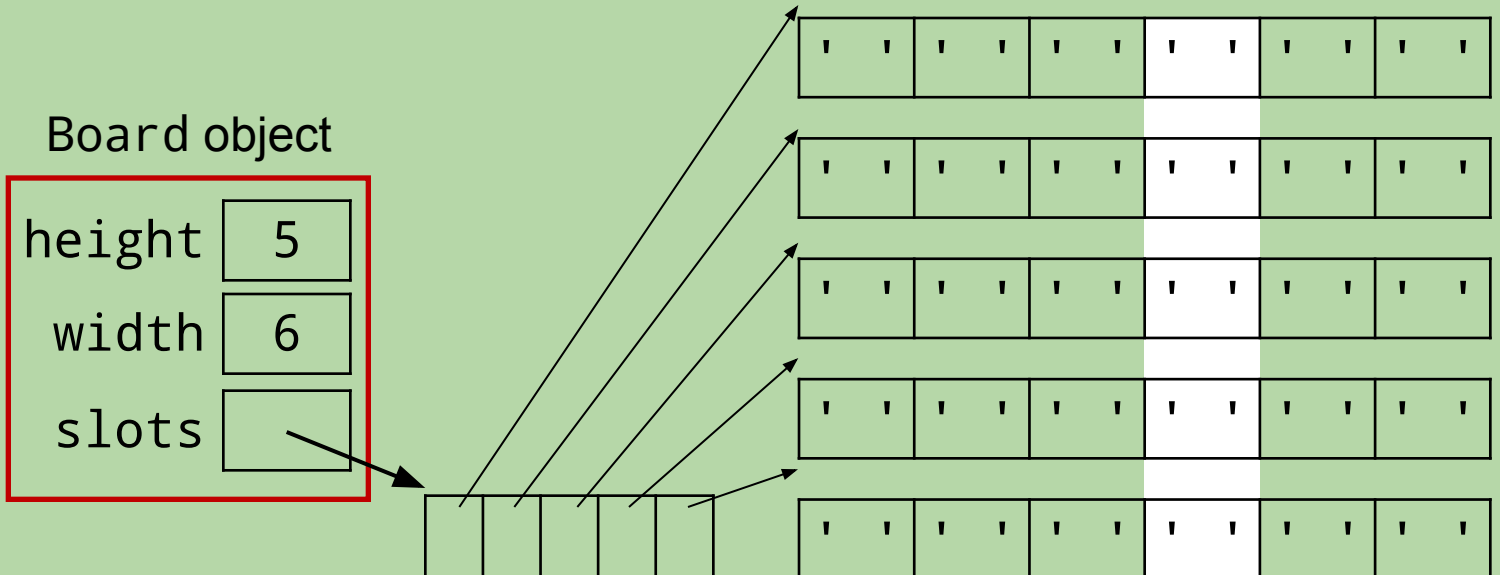
E. `b.slots[2][1] = 'X'`

How would you write this assignment if it were inside a `Board` method?

```
self.slots[2][1] = 'X'
```

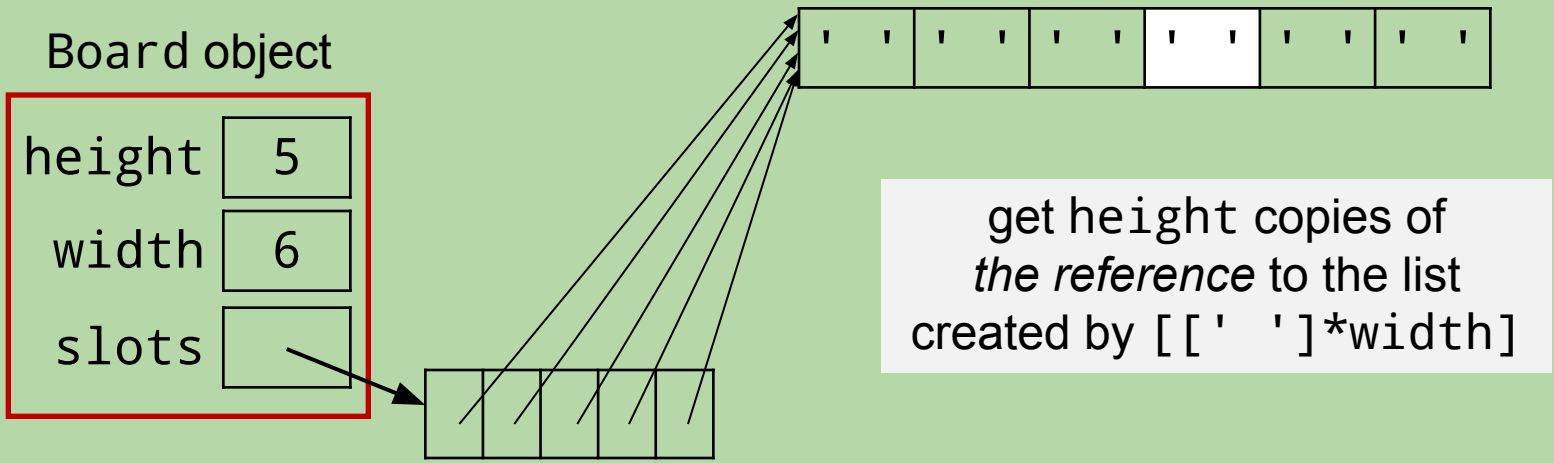
Board Constructor

```
class Board:
    """ a data type for a Connect Four board with
        arbitrary dimensions
    """
    def __init__(self, height, width):
        """ a constructor for Board objects """
        self.height = height
        self.width = width
        self.slots = [[' ']*width] * height  # okay?
```



Board Constructor

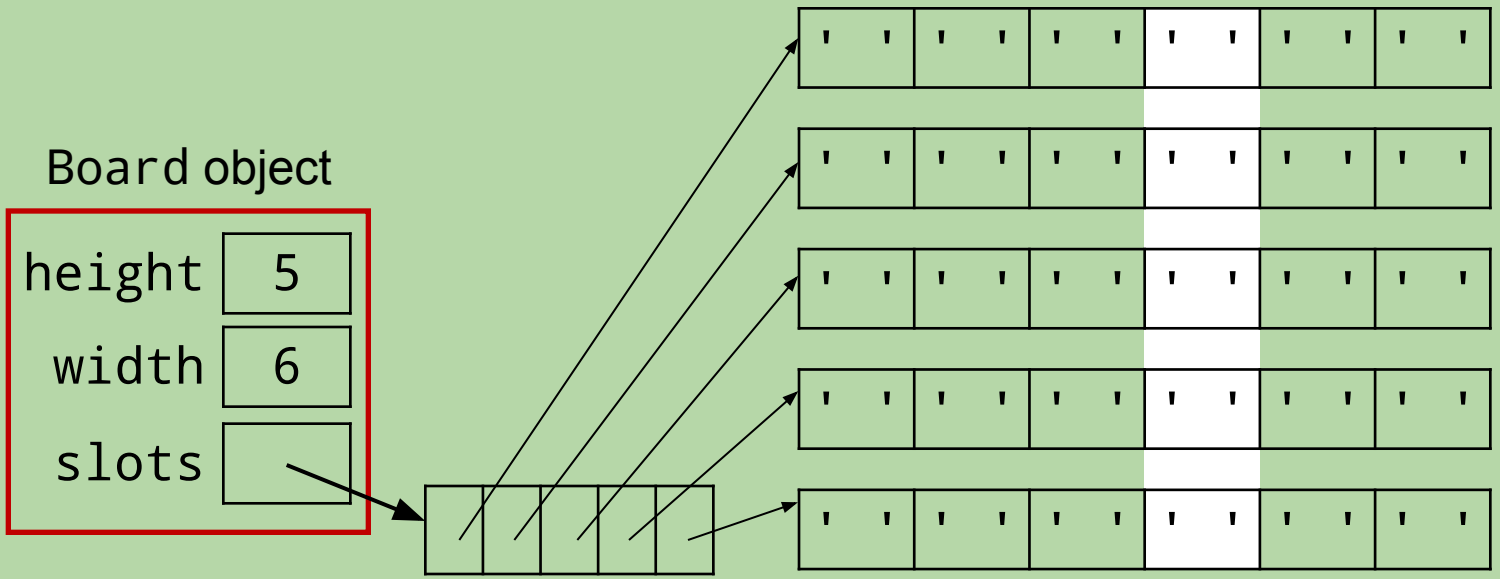
```
class Board:
    """ a data type for a Connect Four board with
        arbitrary dimensions
    """
    def __init__(self, height, width):
        """ a constructor for Board objects """
        self.height = height
        self.width = width
        self.slots = [[' ']*width] * height  # okay? no!
```



Board Constructor

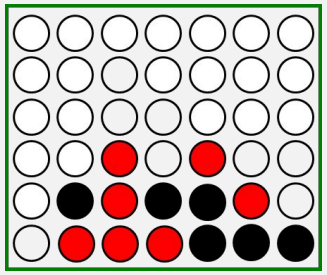
```
class Board:
    """ a data type for a Connect Four board with
        arbitrary dimensions
    """
    def __init__(self, height, width):
        """ a constructor for Board objects """
        self.height = height
        self.width = width
        self.slots = [[' ']*width for r in range(height)]
```

a list comprehension!



__repr__ Method

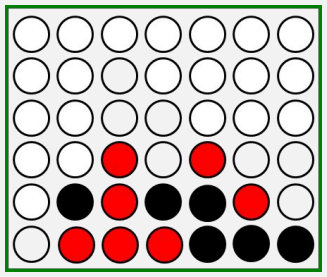
```
def __repr__(self):
    """ returns a string representation of a Board """
    s = ''   # begin with an empty string

    for row in range(self.height):
        s += '|'
        for col in range(self.width):
            s += self.slots[row][col] + '|'
        s += '\n'
    # add the row of hyphens to s
    # add the column indices to s

    return s
```



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

__repr__ Method

```
def __repr__(self):
    """ returns a string representation of a Board """
    s = ''   # begin with an empty string

    for row in range(self.height):
        s += '|'
        for col in range(self.width):
            s += self.slots[row][col] + '|'
        s += '\n'
    s += '--'*self.width    # add the bottom of the board
    s += '-\n'
    for col in range( self.width ):
        s += ' ' + str(col%10)
    s += '\n'

return s
```



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

add_checker(self, checker, col)

```
>>> b = Board(3, 5)
>>> b

| | | | | |
| | | | | |
| | | | | |
-----------
 0 1 2 3 4

>>> b.add_checker('X', 2)
>>> b

| | | | | |
| | | | | |
| | |X| | |
-----------
  0 1 2 3 4
```

```
>>> b.add_checker('O', 4)
>>> b.add_checker('X', 2)
>>> b

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

Which call(s) does the method *get wrong*?

```
class Board:
    ...
    def add_checker(self, checker, col):    # buggy version!
        """ adds the specified checker to column col
            of the called Board object
        """
        row = 0
        while self.slots[row][col] == ' ':
            row += 1
        self.slots[row][col] = checker
```

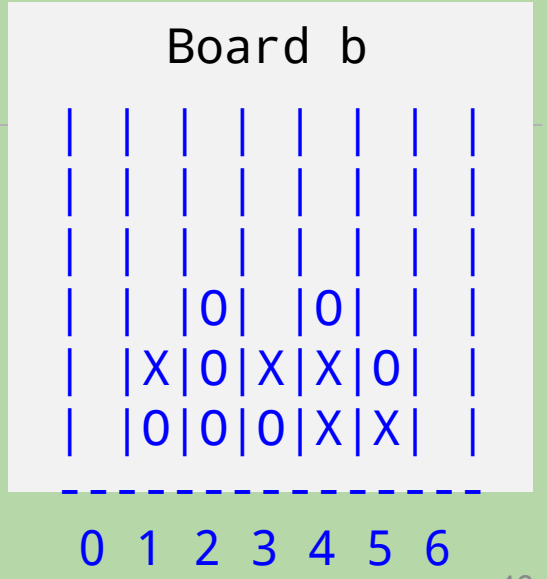
A. `b.add_checker('X', 0)`

B. `b.add_checker('O', 6)`

C. `b.add_checker('X', 2)`

D. A and B

E. A, B, and C



Which call(s) does the method *get wrong*?

```
class Board:
    ...
    def add_checker(self, checker, col):    # buggy version!
        """ adds the specified checker to column col
            of the called Board object
        """
        row = 0
        while self.slots[row][col] == ' ':
            row += 1
        self.slots[row][col] = checker
```

A. `b.add_checker('X', 0)`

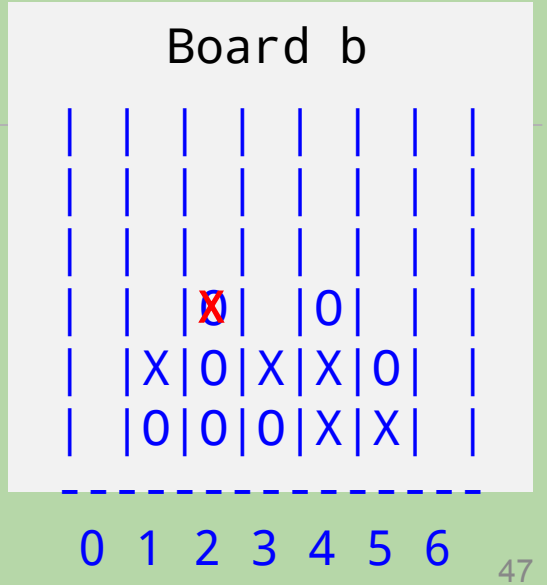
B. `b.add_checker('O', 6)`

IndexError: go past bottom row

C. `b.add_checker('X', 2)` — changes wrong slot

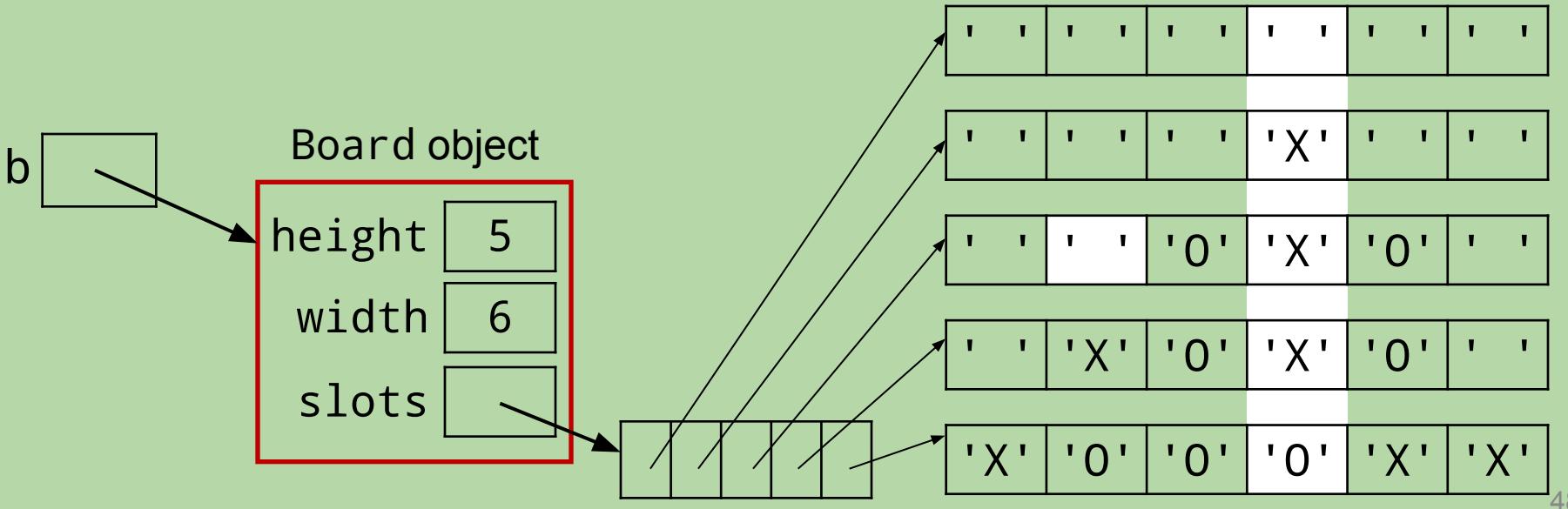
D. A and B

E. A, B, and C



Board Class for Connect Four

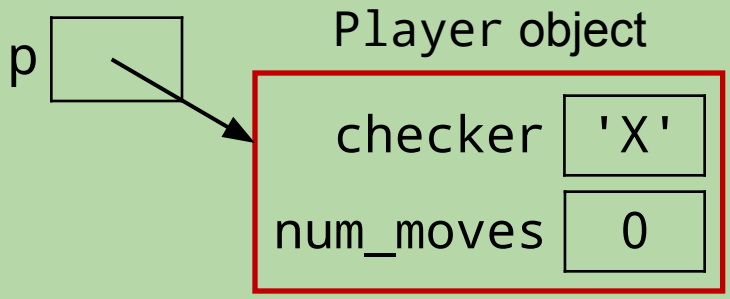
```
class Board:
    def __init__(self, height, width):
        ...
    def __repr__(self):
        ...
    def add_checker(self, checker, col):
        ...
    # plus other methods!
```



Player Class

```
class Player:
    def __init__(self, checker):
        ...
    def __repr__(self):
        ...
    def opponent_checker(self):
        ...
    def next_move(self, board):
        self.num_moves += 1
        while True:
            col = int(input('Enter a column: '))
            # if valid column index, return that integer
            # else, print 'Try again!' and keep looping
```

p = Player('X')



The APIs of Our `Board` and `Player` Classes

```
class Board:
    __init__(self,col)
    __repr__(self)
    add_checker(self,checker,col)
    clear(self)
    add_checkers(self,colnums)
    can_add_to(self,col)
    is_full(self)
    remove_checker(self,col)
    is_win_for(self,checker)

class Player:
    __init__(self,col)
    __repr__(self)
    opponent_checker(self)
    next_move(self,board)
```

Make sure to take full advantage of these methods in your work on hw06!

What are the appropriate method calls?

```
class Board:
    __init__(self,col)
    __repr__(self)
    add_checker(self,checker,col)
    clear(self)
    add_checkers(self,colnums)
    can_add_to(self,col)
    is_full(self)
    remove_checker(self,col)
    is_win_for(self,checker)

class Player:
    __init__(self,col)
    __repr__(self)
    opponent_checker(self)
    next_move(self,board)
```

```
# client code
def process_move(player,board):
    ...
    # get move from player
    col = ____________________

    # apply the move
    ____________________
    ...
```

What are the appropriate method calls?

```
class Board:
    __init__(self,col)
    __repr__(self)
    add_checker(self,checker,col)
    clear(self)
    add_checkers(self,colnums)
    can_add_to(self,col)
    is_full(self)
    remove_checker(self,col)
    is_win_for(self,checker)

class Player:
    __init__(self,col)
    __repr__(self)
    opponent_checker(self)
    next_move(self,board)
```

```
# client code
def process_move(player,board):
    ...
    # get move from player
    col = player.next_move(board)

    # apply the move
    board.add_checker(..., col)
    ...
```

RandomPlayer, AIPlayer Class

```
class Player:
    def __init__(self, checker):
        ...

    def __repr__(self):
        ...

    def opponent_checker(self):
        ...

    def next_move(self, board):
        self.num_moves += 1
```

???

p = Player('X')