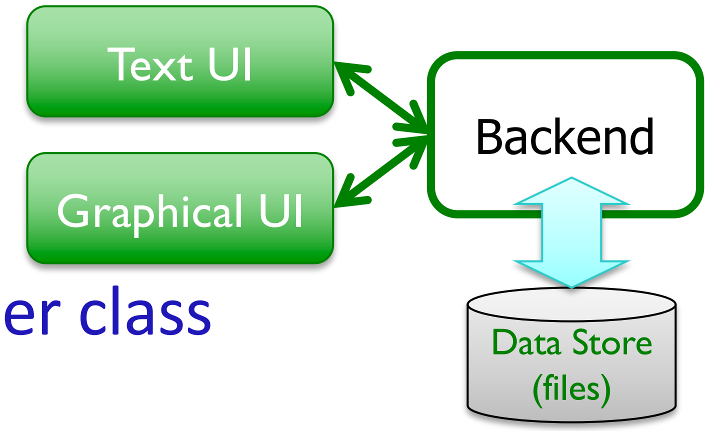
# Reviewing Lab 10



- Created two classes
  - Used one class within another class
  - Tested them
  - Example of a backend to a **real** application
    - Could add a different user interface
- “Good judgment comes from experience”
  - Test methods after writing method
  - Remember your data types
  - Refer to the data type’s API

## Lab 10 Feedback

- Problem solving bonanza!
  - Solving lots of different small problems in a variety of ways
- Use methods you've already written
  - Example: use `addPerson` in `addPeople`
  - Who has this functionality? Do I have access to that object in this method?
- Adhere to interface
  - Accepted parameter types
  - Type of what is returned

## Lab 11: Three Parts

Get out your handouts on binary search (and others)

- Linux practice:
  - Using the **wc** command
- Social Network extensions
  - Exception handling
  - Binary search – find people with a certain name
  - UI: add search functionality
- Two-dimensional lists
  - Including Connect Four

## `wc` Command

- `wc`: Word Count
  - Used to count
    - The lines of Social Network code from Lab 10
    - The lines of code for the whole semester
- Example:
  - `wc –l ../lab10/*.py`
- Specific directions are in the lab

## Searching Our Social Network

In InstaFace, we want to find *person* who has a certain name.

Consider what happens when **`searchlist`** is a list of *Persons* and key is a name (a `str`)

We want to find a `Person` whose name matches the key and return the *Person*

## Binary Search Implementation

```python
def search(searchlist, key):
   low=0
   high = len(searchlist)-1
   while low <= high :
      mid = (low+high)//2
      if searchlist[mid] == key:
         return mid
      elif key > searchlist[mid]:
         # look in upper half
         low = mid+1
      else:
         # look in lower half
         high = mid-1
   return -1
```

## List of Person objects

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Person<br>Id:“1”<br>“Gal” | Person<br>Id:“2”<br>“Scarlett” | Person<br>Id:“3”<br>“Tom” | Person<br>Id: “4”<br>“Ben” | Person<br>Id: “5”<br>“Samuel” |

Example: looking for a person with the name “Tom”…

# List of Person objects

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Person Id:“1” “Gal” | Person Id:“2” “Scarlett” | Person Id:“3” “Tom” | Person Id: “4” “Ben” | Person Id: “5” “Samuel” |

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Person Id: “4” “Ben” | Person Id: “1” “Gal” | Person Id:“5” “Samuel” | Person Id:“2” “Scarlett” | Person Id:“3” “Tom” |

## Extensions to Solution

```
def search(searchlist, key):
   low=0
   high = len(searchlist)-1
   while low <= high :
      mid = (low+high)//2
      if searchlist[mid] == key:
         return mid
      elif key > searchlist[mid]:
         # look in upper half
         low = mid+1
      else:
         # look in lower half
         high = mid-1
   return -1
```

Consider what happens when **`searchlist`** is a list of *Persons* (sorted by name)**,**
***`key`*** is a ***`str`*** representing a name
**Goal**: return a Person object with that name (key)

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Person Id: "4" "Ben" | Person Id: "1" "Gal" | Person Id:"5" "Samuel" | Person Id:"2" "Scarlett" | Person Id:"3" "Tom" |

# Extensions to Solution

```
def search(searchlist, key):
   low=0
   high = len(searchlist)-1
   while low <= high :
      mid = (low+high)//2
      if searchlist[mid] == key:
         return mid
      elif key > searchlist[mid]:
         # look in upper half
         low = mid+1
      else:
         # look in lower half
         high = mid-1
   return -1
```

Consider what happens when **`searchlist`** is a list of *Persons* (sorted by name)**,**
***`key`*** is a ***`str`*** representing a name
**Goal**: return a Person object with that name (key)

What should we do to make search results more intuitive?

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Person Id: "4" "Ben" | Person Id: "1" "Gal" | Person Id:"5" "Samuel" | Person Id:"2" "Scarlett" | Person Id:"3" "Tom" |

## Social Network Searching Overview

- Allows you to search for people by their name–lowercased—for more intuitive results
- Update `Person` and `SocialNetwork` classes and UI appropriately
  - Specific directions are in the lab

## Summary of Search Additions

- Add a search method to `SocialNetwork` class
    - Takes as a parameter the name to search for
        - Need to *lowercase* that name for more intuitive results
    - Original binary search function took a list as a parameter; our method does not
        - Where should we get our list to search?
    - The list to search must be sorted in alphabetical order by name, lowercased
- Check the *name* of the Person that is at the midpoint, lowercased
    - If they match, return that Person
    - Otherwise, …
- Represent (in method) and handle (in UI) when no person has that name

## SocialNetwork Code

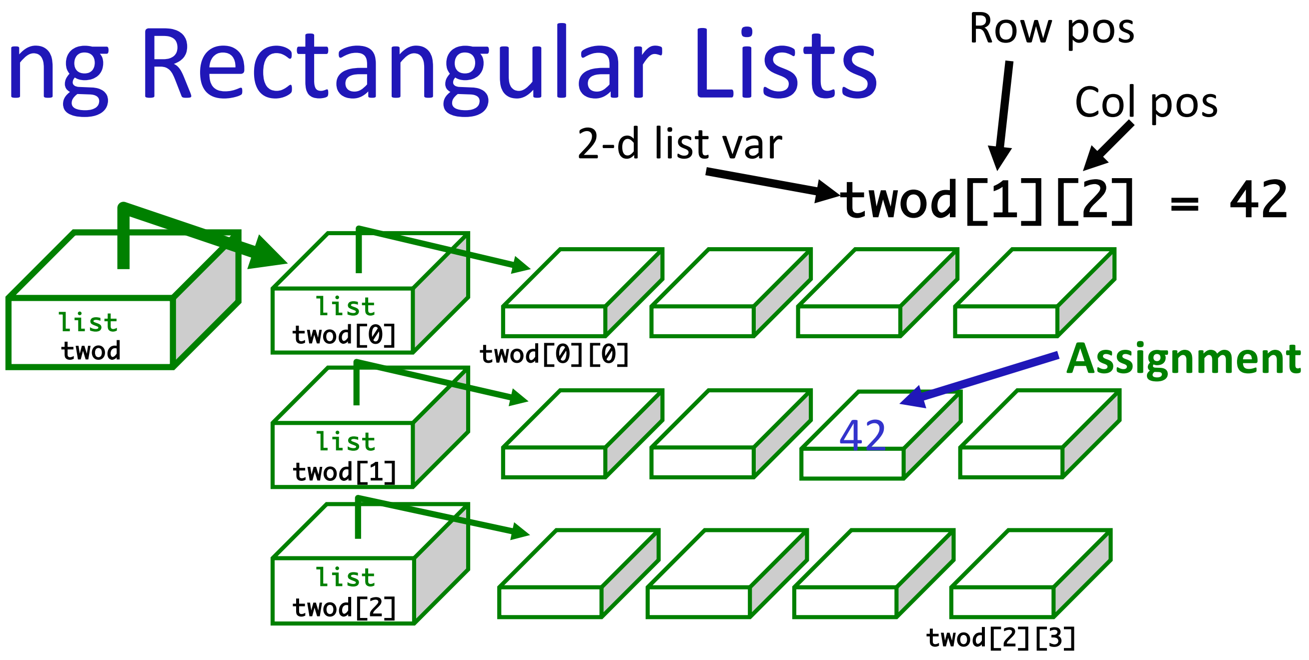
- Fix the major problems in your code first
- Or, use the code in the `lab10_solution` directory
    - `person.py, social.py, instaface.py`

## 2D LISTS

## Review

- How do you create a 2D list?
- How do you get the 2nd element in the 3rd “row” of a list?
- How do you find the number of lists in a 2D list?
- How do you find the number of elements in one of those lists?
- What was tricky about how csplot displays 2D lists?

# Handling Rectangular Lists



- What does each component of `twod[1][2]` mean?
- How many rows does `twod` have, in general?
  - `rows = len(twod)`
- How many columns does `twod` have, in general?
  - `cols = len(twod[0])`

## Game Board for Connect Four

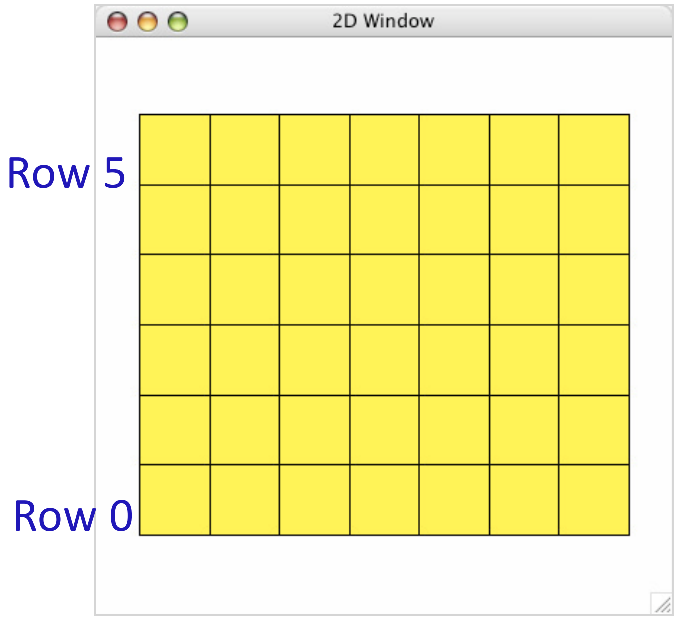
- 6 rows, 7 columns board
- Players alternate dropping red/black checker into slot/column
- Player wins when have four checkers in a row vertically, horizontally, or diagonally

How do we represent the board as a 2D list, using a graphical representation?

# Representing Connect Four Game Board

- Using a 2D list: `_board`
  - 6 rows, 7 columns
  - Initially, the board is full of 0s (FREE spots)

| Number | Meaning | Color |
|---|---|---|
| 0 | Free | Yellow |
| 1 | Player 1 | Red |
| 2 | Player 2 | Black |



## ConnectFour Class

- Data
  - Constants
  - Board: _board
    - 6 rows, 7 columns
    - All spaces FREE to start

- Methods
  - Constructor
  - Display the board
  - Play the game
  - Get input/move from user
  - Check if valid move
  - Make move
  - Check if win

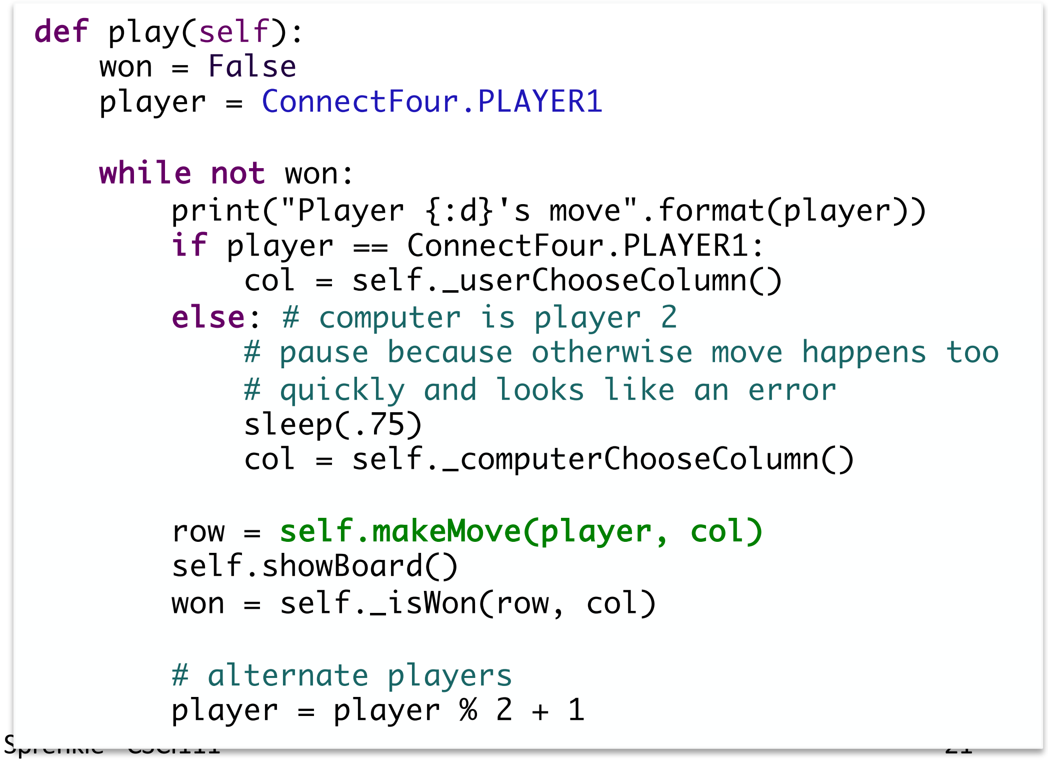
## ConnectFour Constants

```python
class ConnectFour:
    """ Class representing the game Connect Four. """

    # Represent different values on the board
    FREE = 0
    PLAYER1 = 1
    PLAYER2 = 2

    # Represent the dimensions of the board
    ROWS = 6
    COLS = 7
```

To reference constants, use `ConnectFour.CONSTANT`

## ConnectFour Class

- Implementation of method to play the game
  - Repeat:
    - Get input/move from user (depending on whose turn it is)
    - Make move
    - Display board
    - Check if win
    - Change player



```python
def play(self):
    won = False
    player = ConnectFour.PLAYER1

    while not won:
        print("Player {:d}'s move".format(player))
        if player == ConnectFour.PLAYER1:
            col = self._userChooseColumn()
        else: # computer is player 2
            # pause because otherwise move happens too
            # quickly and looks like an error
            sleep(.75)
            col = self._computerChooseColumn()

        row = self.makeMove(player, col)
        self.showBoard()
        won = self._isWon(row, col)

        # alternate players
        player = player % 2 + 1
```

## Connect Four (C4): Making moves

- Precondition: User selects a valid column
- Postcondition: “Checker” is filled in at that column

Enforcement of precondition:

```
def _userChooseColumn(self):
    # gets the column where user clicked
    col = csplot.sqinput()
    validMove = self._isValidMove(col)
    while not validMove:
        print("NOT A VALID MOVE.")
        print("PLEASE SELECT AGAIN.")
        print()
        col = csplot.sqinput()
        validMove = self._isValidMove(col)
    return col
```

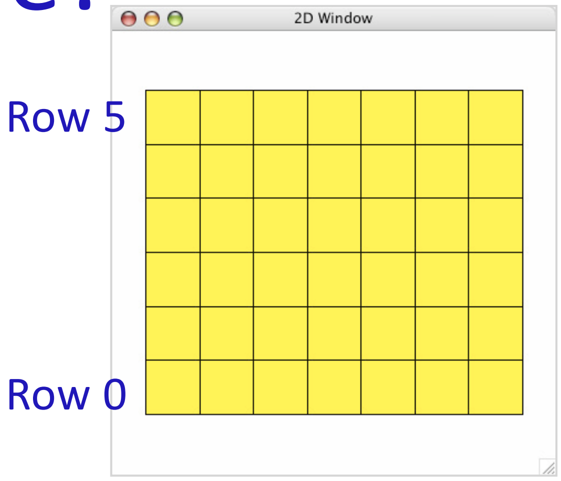
## Problem: C4 - Valid move?

- Need to enforce valid moves
  - In physical game, run out of spaces for checkers if not a valid move

- How can we determine if a move is valid?
  - How do we know when a move is ***not*** valid?

# Problem: C4 - Valid move?

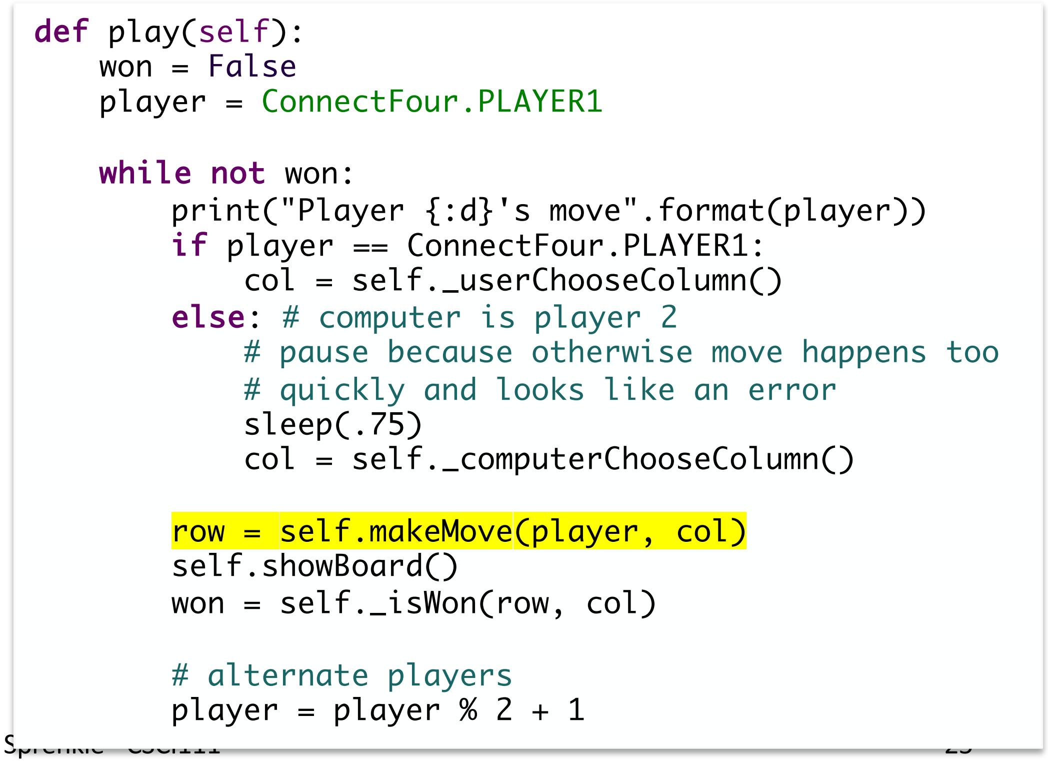
- Solution: check the “top” spot
  - If the spot is FREE, then it’s a valid move



```
def _isValidMove(self, col):
    """
    Return True iff the dropping a checker in this col (an int)
    represents a valid move.
    """
    return self._board[ConnectFour.ROWS-1][col] == ConnectFour.FREE
```

## ConnectFour Class

- Implementation of play the game method
  - Repeat:
    - Get input/move from user (depending on whose turn it is)
    - Make move
    - Display board
    - Check if win
    - Change player



```python
def play(self):
    won = False
    player = ConnectFour.PLAYER1

    while not won:
        print("Player {:d}'s move".format(player))
        if player == ConnectFour.PLAYER1:
            col = self._userChooseColumn()
        else: # computer is player 2
            # pause because otherwise move happens too
            # quickly and looks like an error
            sleep(.75)
            col = self._computerChooseColumn()

        row = self.makeMove(player, col)
        self.showBoard()
        won = self._isWon(row, col)

        # alternate players
        player = player % 2 + 1
```

## Problem: C4 - Making a Move

- Given: a column for where the "checker" goes; which player made the move
- Precondition: Valid column
- Postcondition: "Checker" is filled in at that column; the row where the checker "lands" is returned

How do we implement this method?

## Lab 11 Directory

- To start, your directory should look like
  - connectfour.py
  - csplot.py
  - instaface.py
  - lab10_solution
  - person.py
  - social.py
  - test.py

Thanks to **Tim** and **Ignas** for their help this semester!