# Objectives

- Two-dimensional lists

# Review

- What is *exception handling*?
  - How do we implement it in our code? What is the structure?
  - What are best practices?
- What are the two types of search we discussed?
  - How do they work?
  - How do they compare?
  - What are the tradeoffs between using linear search and binary search?
- Lists (for today's lesson…)
  - How do we find the number of elements in the list?
  - How can we find the value of the third element in the list?

# Review: Handling Exceptions

- Using try/except statements
- Syntax:

```
try:
        <body>
except [<errorType>] :
        <handler>
```

Optional: use this to handle specific error types appropriately

- Example:

```
try:
    age = int(input("Enter your age: "))
    currentyear = int(input("Enter the current year: "))
except:
    print("Error: Your input was not in the correct form.")
    print("Enter integers for your age and the current year")
    sys.exit(1)
```

Typical/normal behavior

Handle exception

yearborn.py

# Review: Best Practices

- Prevent errors as best you can
  - Example: use `if` statements to verify data
- For errors you can't prevent, *handle* them!
  - Example: We can check if a file exists before trying to read it BUT between the check and actually reading the file, the file could be deleted from the system!

# Review: Search Using `in` Review

- Iterates through a list, checking if the element is found
- Known as ***linear search***
- **Implementation:**

```
def linearSearch(searchlist, key):
    for elem in searchlist:
        if elem == key:
            return True
    return False
```

| | | | | |
|---|---|---|---|---|
| **value** | 8 | 5 | 3 | 7 |
| **pos** | 0 | 1 | 2 | 3 |

# Alternative: Like `index` method

- Iterates through *positions* in a list, checking if the element is found
- Still known as ***linear search***
- **Implementation:**

```
def linearSearch(searchlist, key):
    for pos in len(range(searchlist)):
        if searchlist[pos] == key:
            return pos
    return -1
```

# Review: Linear Search

- **Overview**: Iterates through a list, checking if the element is found
- **Benefits:**
  - Works on *any* list
- **Drawbacks**:
  - **Slow**, on average: needs to check each element of list if the element is not in the list

# Review: Binary Search: Eliminate Half the Possibilities

- Repeat until find value (or looked through all values)
  - Guess middle *value* of possibilities
    - (not middle *position*)
  - If match, found!
  - Otherwise, find out too high or too low
  - Modify your possibilities
    - Eliminate the possibilities from your number and higher/lower, as appropriate
- Benefits: faster than linear search
- Drawbacks: requires sorted list

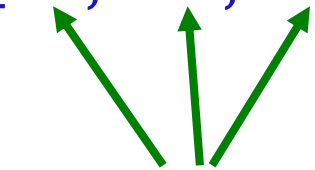
# 2D LISTS

# Lists

- We've used lists that contain
  - Integers
  - Strings
  - Cards (Deck class)
  - Persons (your Person class)
- We discussed that lists can contain multiple types of objects within the same list
  - Wheel of Fortune: ["Bankrupt", 250, 350, …]
- Lists can contain *any* **type** of object
  - Even **LISTS**!

# Review of Regular (1D) Lists

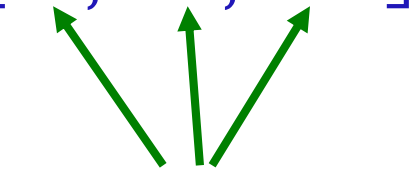
```
onedlist = [ 7, -1, 23 ]
```




- How do we find the number of elements in the list?
- How can we find the value of the third element in the list?

# Review of Regular (1D) Lists

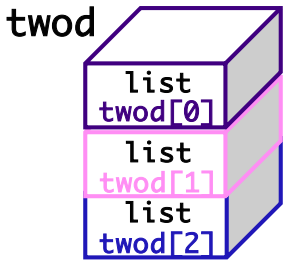
```
onedlist = [ 7, -1, 23 ]
```




- `len(onedlist)` is 3
- `onedlist[2]` is 23

# A List of Lists: 2-Dimensional List

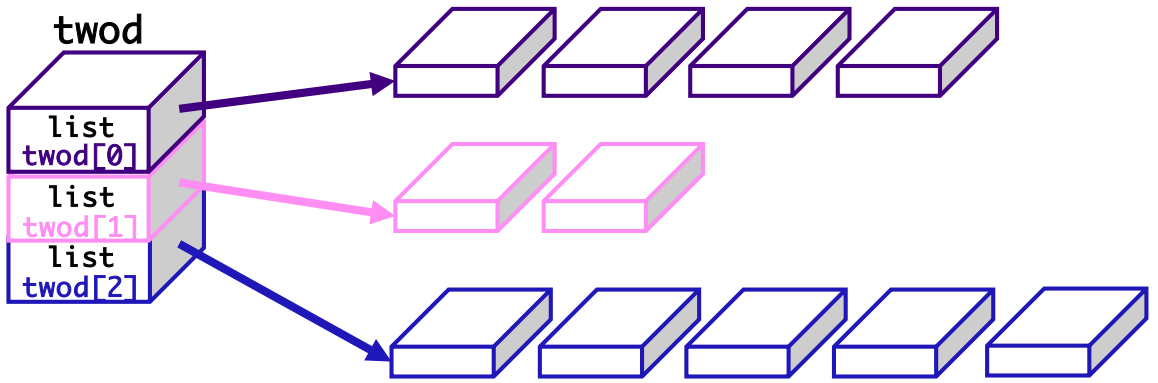
```
                 twod[0]   twod[1]     twod[2]
twod = [ [1,2,3,4], [5,6], [7,8,9,10,11] ]
```




1st dimension

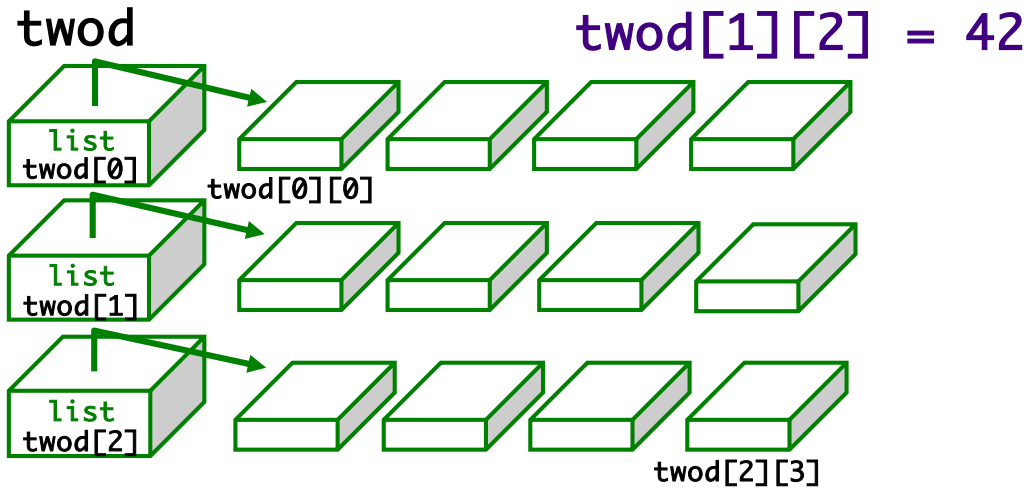
# A List of Lists: 2-Dimensional list

```
twod = [ [1,2,3,4], [5,6], [7,8,9,10,11] ]
```



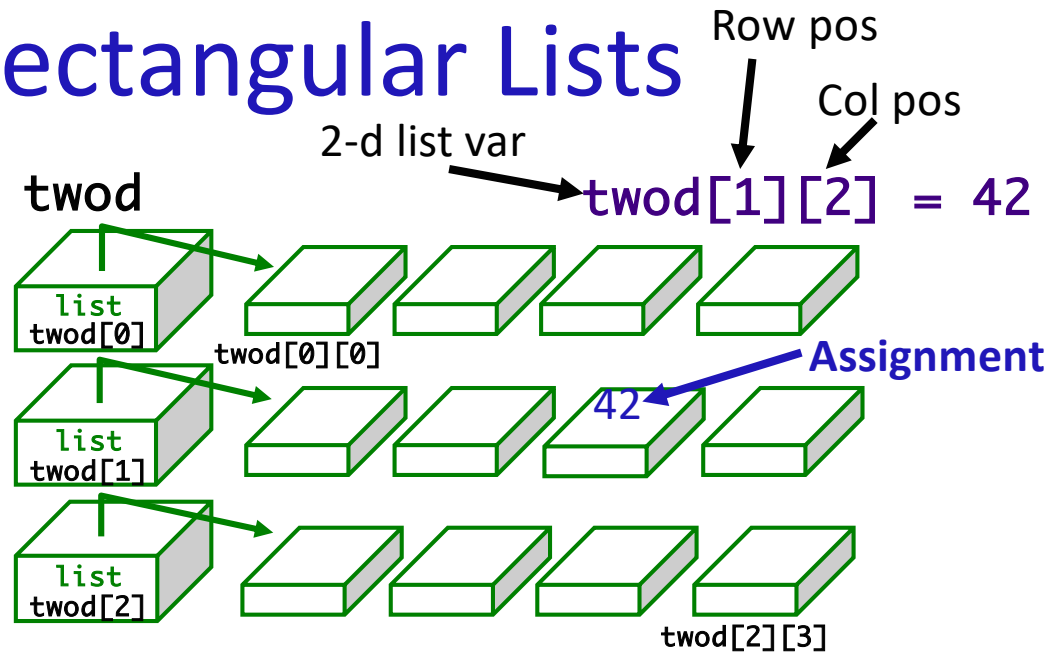

- "Rows" within 2-dimensional list do **not** need to be the same length
- However, it's often easier if they're the same length!
  - We'll focus on "**rectangular**" 2D lists

# Handling Rectangular Lists




- What does each component of `twod[1][2]` mean?
- How can we programmatically determine the number of rows in `twod`? The number of columns in a given row?

# Handling Rectangular Lists

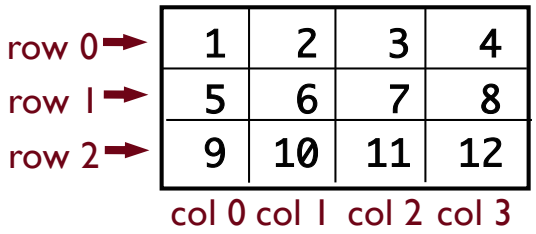



- How can we programmatically determine the number of rows in `twod`?
  - `rows = len(twod)`
- The number of columns in a given row?
  - `cols = len(twod[whichRow])`

# 2D List Practice

Starting with the 2D list **twod** shown, what are the values in **twod** after running this code?

**twod** Before




```
def mystery(twod):
    """ 'run' this on twod, at right """
    for row in range( len(twod) ):
        for col in range( len(twod[row]) ):
            if row == col:
                twod[row][col] = 42
            else:
                twod[row][col] += 1
```

**twod** After

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |

# 2D List Practice

Starting with the 2D list **twod** shown, what are the values in **twod** after running this code?

**twod** Before

| | col 0 | col 1 | col 2 | col 3 |
|---|---|---|---|---|
| row 0 → | 1 | 2 | 3 | 4 |
| row 1 → | 5 | 6 | 7 | 8 |
| row 2 → | 9 | 10 | 11 | 12 |

```
def mystery(twod):
    """ 'run' this on twod, at right """
    for row in range( len(twod) ):
        for col in range( len(twod[row]) ):
            if row == col:
                    twod[row][col] = 42
            else:
                    twod[row][col] += 1
```

**twod** After

| | | | |
|---|---|---|---|
| 42 | 3 | 4 | 5 |
| 6 | 42 | 8 | 9 |
| 10 | 11 | 42 | 13 |

mystery.py

# Example Process for Creating a 2D List

```
twod = [ ]
```

- Create a row of the list, e.g.,

```
row = [1, 2, 3, 4] or row = list(range(1,5))
or row = [0] * 4 or …
```

- Then append that row to the list

```
twod.append( row )
print(twod)
```

  - [ [1, 2, 3, 4] ]

- Repeat

```
row = list(range(1,5))
twod.append( row )
print(twod)
```

  - [ [1, 2, 3, 4], [1, 2, 3, 4] ]

# Generalize Creating a 2D List

- Create a function that returns a 2D list with width ***cols*** and height ***rows***
  - Initialize each element in (sub) list to 0

```
def create2DList(rows, cols):
```

# Generalize Creating a 2D List

- Create a function that returns a 2D list with width ***cols*** and height ***rows***
  - Initialize each element in (sub) list to 0

```
def create2DList(rows, cols):
    twodlist = [ ]
    # for each row
    for rowPos in range( rows ):
        row = [ ]
        # for each column, in each row
        for colPos in range( cols ):
            row.append(0)
        twodlist.append(row)
    return twodlist
```

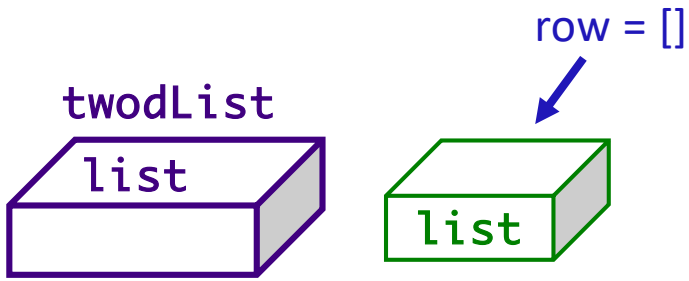
# Generalize Creating a 2D List

- Create a function that returns a 2D list with width ***cols*** and height ***rows***
  - Initialize each element in (sub) list to 0

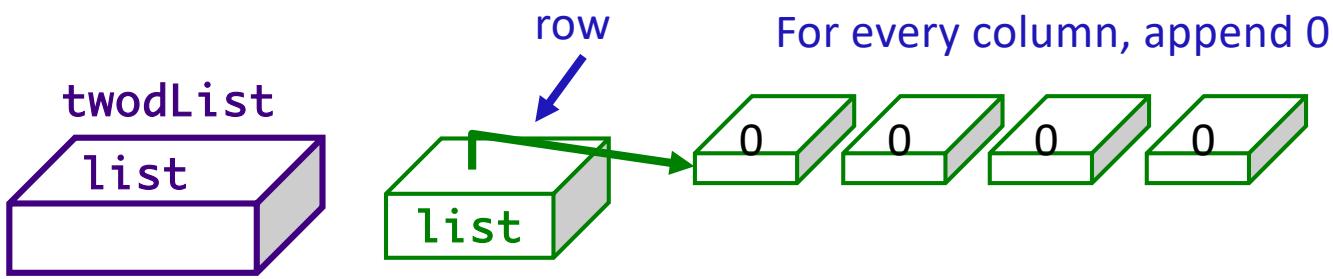
```
def create2DList(rows, cols):
    twodlist = [ ]
    # for each row
    for rowPos in range( rows ):
        row = [ ]
        # for each column, in each row
        for colPos in range( cols ):
            row.append(0)
        twodlist.append(row)
    return twodlist
```

Change here for different elements added into the list

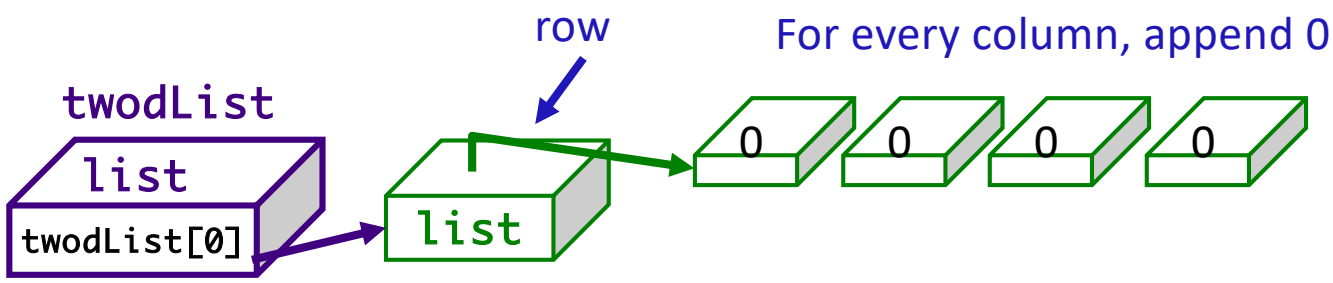
# Example: Creating 2D List – 3 rows, 4 cols



# Example: Creating 2D List – 3 rows, 4 cols

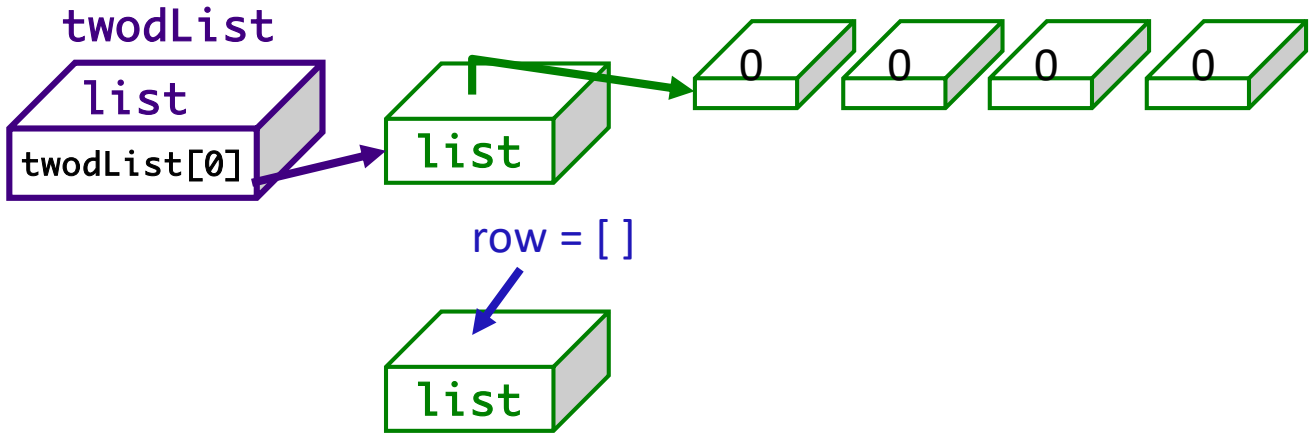


# Example: Creating 2D List – 3 rows, 4 cols

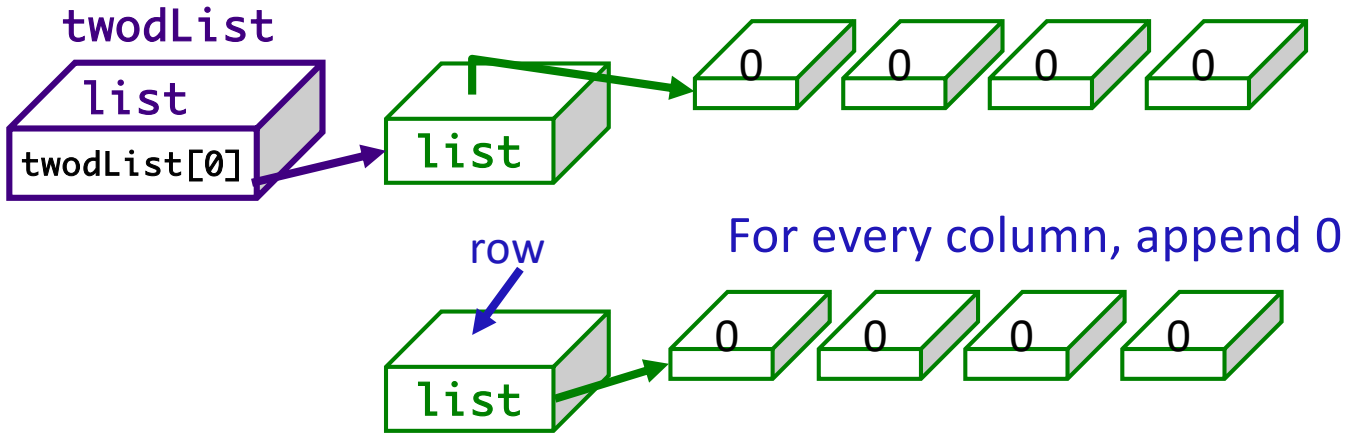



Append `row` to `twodlist`

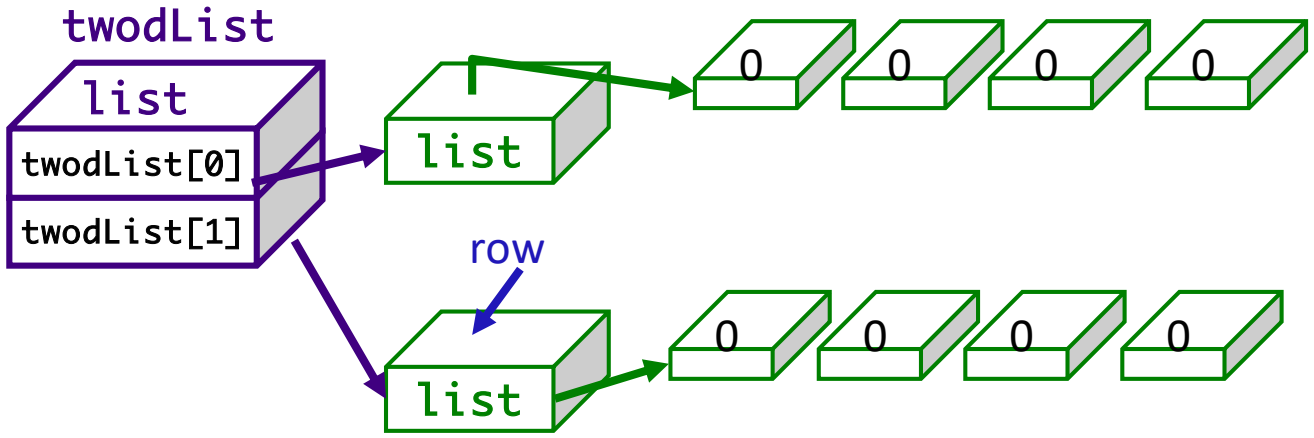
# Example: Creating 2D List – 3 rows, 4 cols



# Example: Creating 2D List – 3 rows, 4 cols

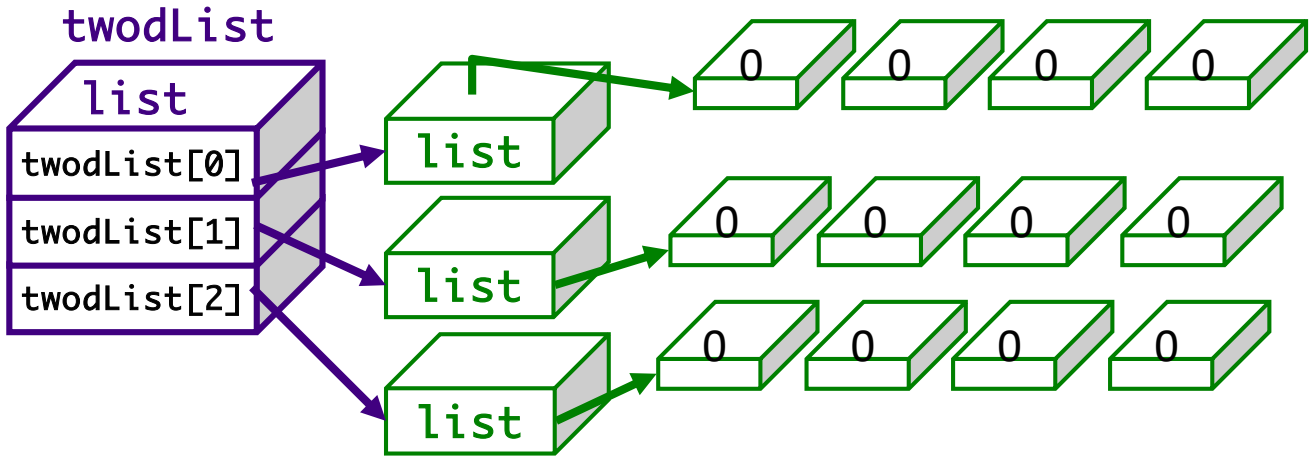


# Example: Creating 2D List – 3 rows, 4 cols




Append `row` to `twodList`

# Example: Creating 2D List – 3 rows, 4 cols



# Generalize Creating a 2D List

- Create a function that returns a 2D list with width ***cols*** and height ***rows***
  - Initialize each element in (sub) list to 0

```
def create2DList(rows, cols):
    twodlist = [ ]
    # for each row
    for rowPos in range( rows ):
        row = [ ]
        # for each column, in each row
        for colPos in range( cols ):
            row.append(0)
        twodlist.append(row)
    return twodlist
```

Change here for different elements added into the list

# Incorrect: Creating a 2D List

- The following code **won't** work. Why?

```
def noCreate2DList(rows, cols):
    twodlist = [ ]
    row = [ ]

    for col in range( cols ):
        row.append(0)

    for r in range( rows ):
        twodlist.append(row)
    return twodlist
```

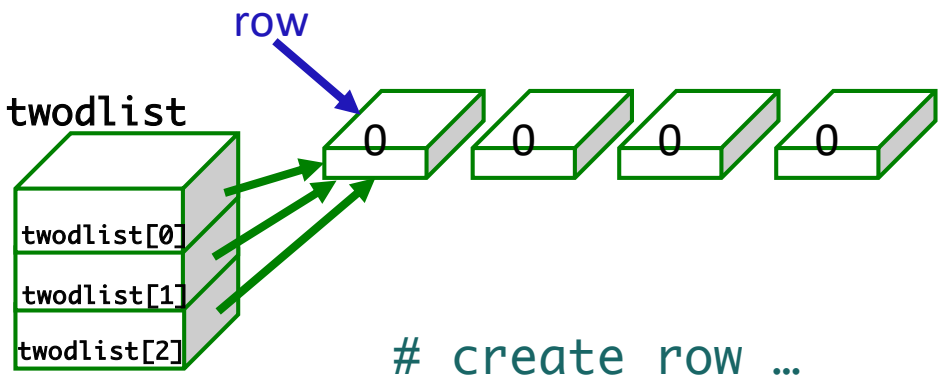
Explain this output from using this [incorrect] function to create a 2D list

```
Incorrect Matrix Creation:
------------------------------
Matrix:
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]

Assigning matrix[1][2] = 3

Result:
[[0, 0, 3, 0], [0, 0, 3, 0], [0, 0, 3, 0]]
```

twod_exercises.py

# All Rows of 2D List Point at Same Block of Memory

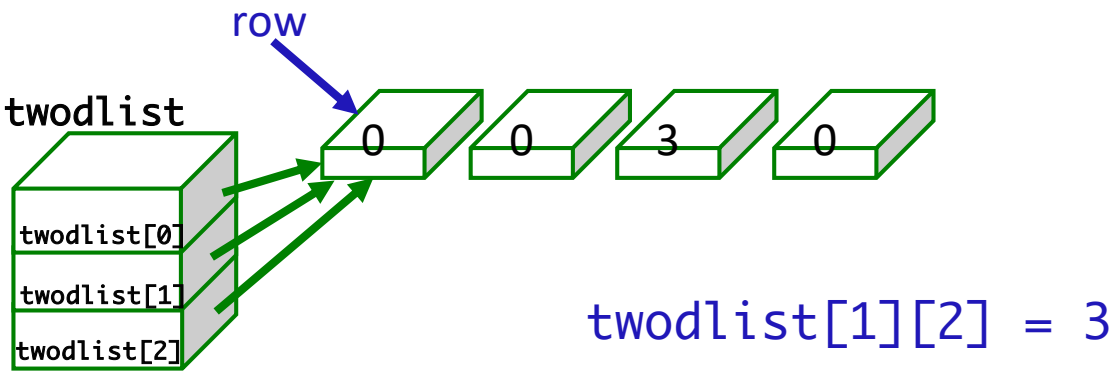
- Each “row” points to the **same** list in memory




```
# create row …
twodlist.append(row)
twodlist.append(row)
twodlist.append(row)
```

# All Rows of 2D List Point at Same Block of Memory

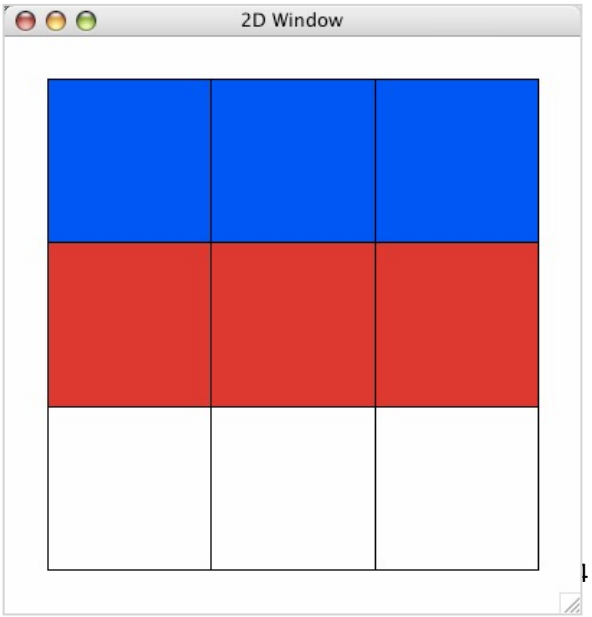
- Each “row” points to the **same** list in memory



# Graphical Representation of 2D Lists

- Module: `csplot`
- Allows you to visualize your 2D list
  - Numbers are represented by different colors

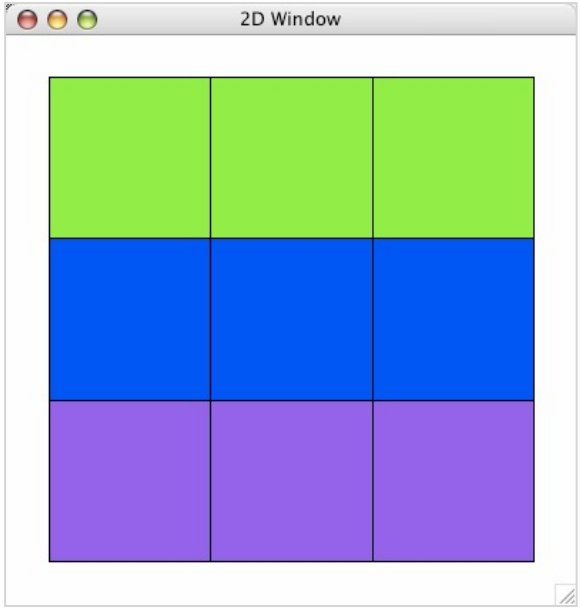
```
import csplot
…
# create 2D list…
twodlist=[ [0,0,0], [1,1,1], [2,2,2] ]
# display list graphically
csplot.show(twodlist)
```



# Graphical Representation of 2D Lists

- Can map colors to numbers

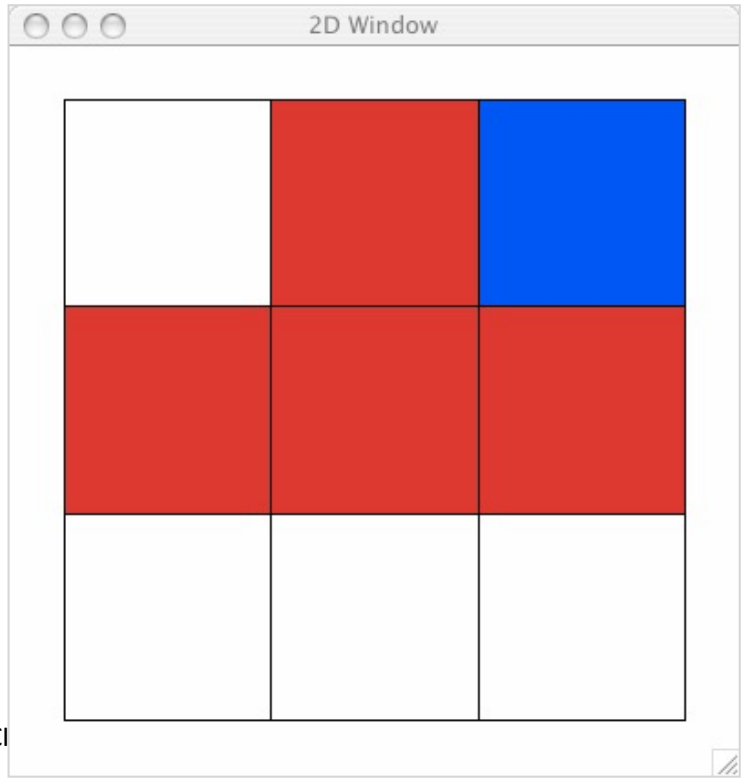
```
import csplot
…
# create 2D list…
twodlist=[ [0,0,0], [1,1,1], [2,2,2] ]
# create dictionary of numbers to color rep
numToColor = {0:"purple", 1:"blue", 2:"green"}
csplot.show(twodlist, numToColor)
```



# Graphical Representation of 2D Lists

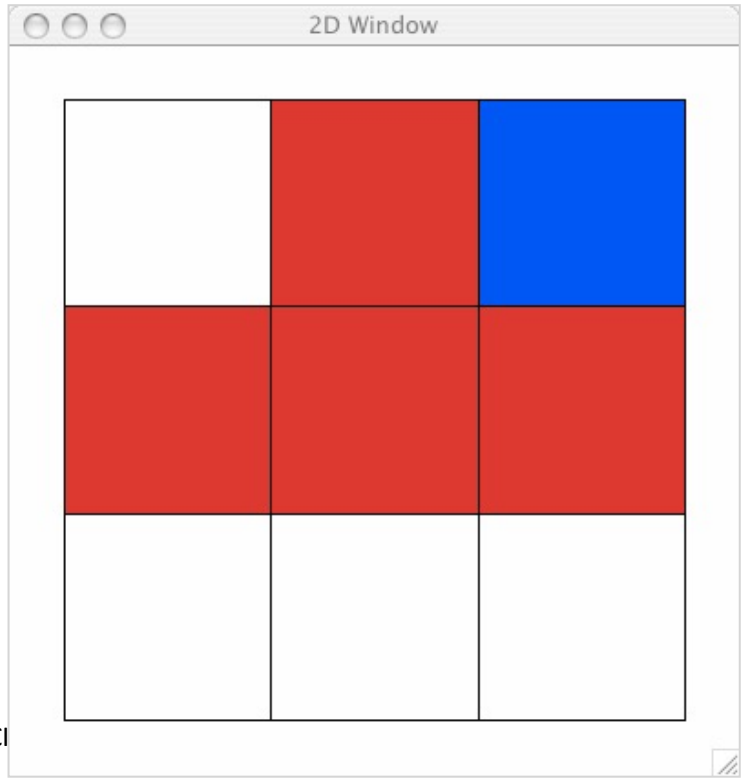
```
matrix = [[0,0,0], [1,1,1], [0,1,2]]
```

What values map to
which colors by default?
Other observations?



# Graphical Representation of 2D Lists

```
matrix = [[0,0,0], [1,1,1], [0,1,2]]
```




What values map to which colors by default?

- Note representation of list/rows is not how we've been visualizing

# 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

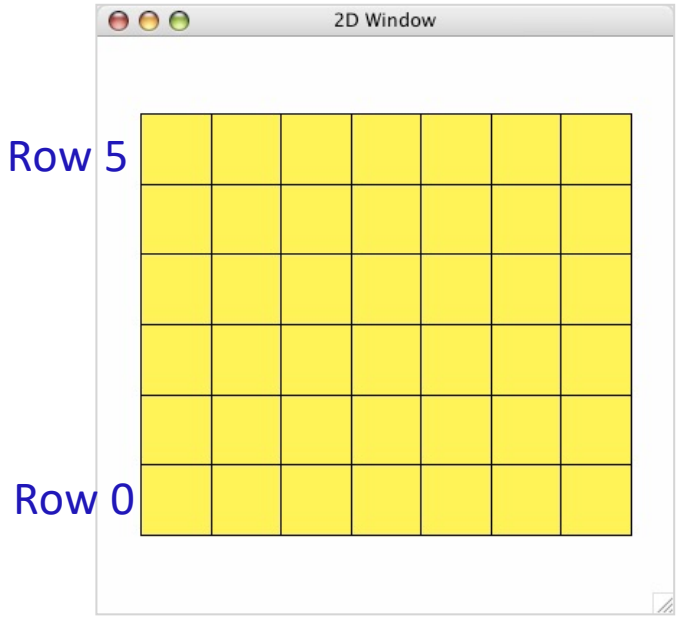
- Using a 2D list

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

# Representing Connect Four Game Board

- Using a 2D list

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



# ConnectFour Class

- What is the data associated with the class?
- What methods should we implement?

# ConnectFour Class

- Data
  - Constants
  - 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

```
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 class's constants, use `ConnectFour.CONSTANT`

# 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

```
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

- User clicks on a column
  - "Checker" is filled in at that column

```
# gets the column where user clicked
col = csplot.sqinput()
```

```
    def _userChooseColumn(self):
        """Allow the user to pick a column."""
        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

# Problem: C4 - Making a Move

- The player clicks on a column, meaning that's where the player wants to put a checker
- How do we update the board?

# Looking Ahead

- Lab 11 – Tomorrow
  - Pre lab: Exception Handling
    - review nested lists, classes
  - Review implementation of binary search
- Broader Issue Friday