2-D Lists;
References Revisited

Computer Science 111
Boston University

Vahid Azadeh-Ranjbar, Ph.D.

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

2-D Lists

- Recall that a list can include sublists

```
mylist = [17, 2, [2, 5], [1, 3, 7]]
```

  - what is `len(mylist)`?

2-D Lists

- Recall that a list can include sublists

```
mylist = [17, 2, [2, 5], [1, 3, 7]]
```

  - what is `len(mylist)`? 4

- To capture a rectangular table or grid of values, use a *two-dimensional* list:

```
table = [[15,  8,  3, 16, 12,  7,  9   5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

  - a list of sublists, each with the same length
  - each sublist is one "row" of the table

2-D Lists: Try These Questions!

```
table = [[15,  8,  3, 16, 12,  7,  9,  5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

- what is `len(table)`?
- what does `table[0]` represent?

  `table[1]`?

  `table[-1]`?
- what is `len(table[0])`?
- what is `table[3][1]`?

- how would you change the `1` in the lower-left corner to a 7?

2-D Lists: Try These Questions!

```
table = [[15,  8,  3, 16, 12,  7,  9,  5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

- what is `len(table)`? 5 (more generally, the # of rows / height)
- what does `table[0]` represent? the first/top row

  `table[1]`? the second row

  `table[-1]`? the last/bottom row
- what is `len(table[0])`? 8 (the # of columns / width)
- what is `table[3][1]`? 14

  row index column index
- how would you change the `1` in the lower-left corner to a `7`?

```
table[4][0] = 7       # table[-1][0] = 7 also works!
```

Dimensions of a 2-D List

```
table = [[15,  8,  3, 16, 12,  7,  9,  5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

`len(table)` is the # of rows in `table`

`table[r]` is the row with index `r`

`len(table[r])` is the # of elements in row `r`

`len(table[0])` is the # of columns in `table`

Picturing a 2-D List

```
table = [[15,  8,  3, 16, 12,  7,  9,  5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

- Here's one way to picture the above list:

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	15	8	3	16	12	7	9	5
1	6	11	9	4	1	5	8	13
2	17	3	5	18	10	6	7	21
3	8	14	13	6	13	12	8	4
4	1	9	5	16	20	2	3	9

column indices

row indices

Accessing an Element of a 2-D List

```
table = [[15,  8,  3, 16, 12,  7,  9,  5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

`table[r][c]` is the element at row `r`, column `c` in `table`

examples:

```
>>> print(table[2][1])
3
```

column index

row index

Accessing an Element of a 2-D List

```
table = [[15,  8,  3, 16, 12,  7,  9,  5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  0,  9]]
```

`table[r][c]` is the element at row `r`, column `c` in `table`

examples:

```
>>> print(table[2][1])
3
```

column index

row index

```
>>> table[-1][-2] = 0
```

Using Nested Loops to Process a 2-D List

```
table = [[15,  8,  3, 16, 12,  7,  9,  5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

```
for r in range(len(table)):
    for c in range(len(table[0])):
        # process table[r][c]
```

Using Nested Loops to Process a 2-D List

```
table = [[15, 19,  3, 16],
         [ 6, 21,  9,  4],
         [17,  3,  5, 18]]
count = 0
for r in range(len(table)):
    for c in range(len(table[0])):
        if table[r][c] > 15:
            count += 1
print(count)
```

r	c	table[r][c]	count

Using Nested Loops to Process a 2-D List

```
table = [[15, 19,  3, 16],
         [ 6, 21,  9,  4],
         [17,  3,  5, 18]]
count = 0
for r in range(len(table)):
    for c in range(len(table[0])):
        if table[r][c] > 15:
            count += 1
print(count)                     # prints 5
```

r	c	table[r][c]	count
			0
0	0	15	0
0	1	19	1
0	2	3	1
0	3	16	2
1	0	6	2
1	1	21	3
...			
2	0	17	4
...			
2	3	18	5

Which Of These Counts the Number of Evens?

```
table = [[15, 19,  3, 16],
         [ 6, 21,  9,  4],
         [17,  3,  5, 18]]
```

A.

```
count = 0
for r in range(len(table)):
    for c in range(len(table[0])):
        if table[r][c] % 2 == 0:
            count += 1
```

B.

```
count = 0
for r in len(table):
    for c in len(table[0]):
        if c % 2 == 0:
            count += 1
```

C.

```
count = 0
for r in range(len(table[0])):
    for c in range(len(table)):
        if table[r][c] % 2 == 0:
            count += 1
```

D. either A or B

E. either A or C

Which Of These Counts the Number of Evens?

```
table = [[15, 19,  3, 16],
         [ 6, 21,  9,  4],
         [17,  3,  5, 18]]
```

A.

```
count = 0
for r in range(len(table)):
    for c in range(len(table[0])):
        if table[r][c] % 2 == 0:
            count += 1
```

B.

```
count = 0
for r in len(table):
    for c in len(table[0]):
        if c % 2 == 0:
            count += 1
```

C.

```
count = 0
for r in range(len(table[0])):
    for c in range(len(table)):
        if table[r][c] % 2 == 0:
            count += 1
```

D. either A or B

E. either A or C

Using Nested Loops to Process a 2-D List

```
table = [[15, 19,  3, 16],
         [ 6, 21,  9,  4],
         [17,  3,  5, 18]]
count = 0
for r in range(len(table)):
    for c in range(len(table[0])):
        if table[r][c] % 2 == 0:
            count += 1
print(count)
```

r	c	table[r][c]	count

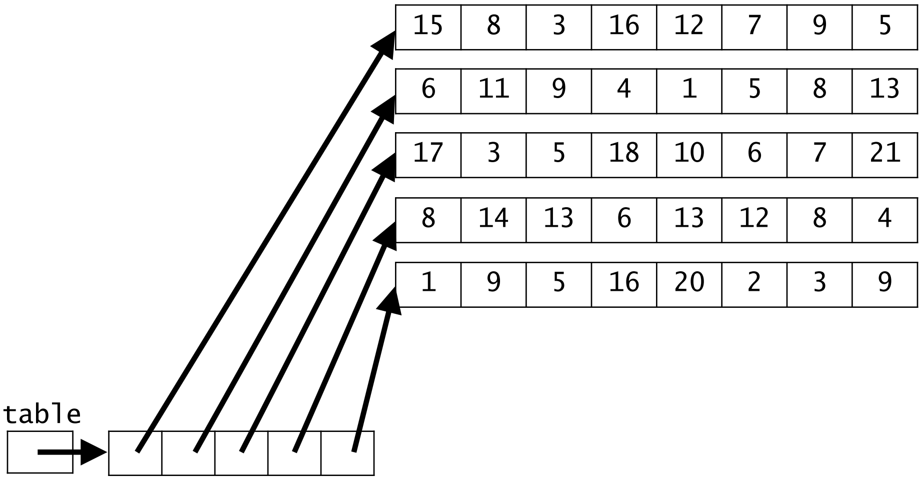
Using Nested Loops to Process a 2-D List

```
table = [[15, 19,  3, 16],
         [ 6, 21,  9,  4],
         [17,  3,  5, 18]]
count = 0
for r in range(len(table)):
    for c in range(len(table[0])):
        if table[r][c] % 2 == 0:
            count += 1
print(count)                  # prints 4
```

r	c	table[r][c]	count
			0
0	0	15	0
0	1	19	0
0	2	3	0
0	3	16	1
1	0	6	2
1	1	21	2
...			
1	3	4	3
...			
2	3	18	4

Picturing a 2-D List (cont)

- Here's a more accurate picture:

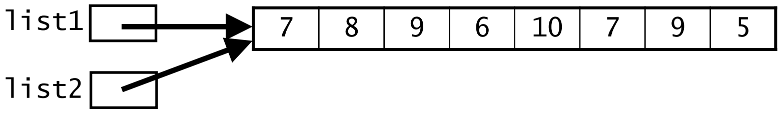


15	8	3	16	12	7	9	5
6	11	9	4	1	5	8	13
17	3	5	18	10	6	7	21
8	14	13	6	13	12	8	4
1	9	5	16	20	2	3	9

Recall: Copying a List

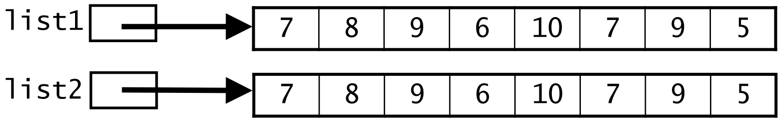
- We can't copy a list by a simple assignment:

```
list1 = [7, 8, 9, 6, 10, 7, 9, 5]
list2 = list1
```



- We can copy this list using a full slice:

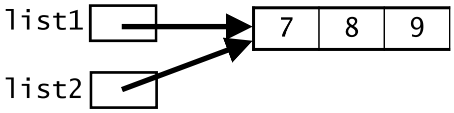
```
list1 = [7, 8, 9, 6, 10, 7, 9, 5]
list2 = list1[:]
```



Changing the Internals vs. Changing a Variable

- When two variables hold a reference to the same list...

```
list1 = [7, 8, 9]
list2 = list1
```

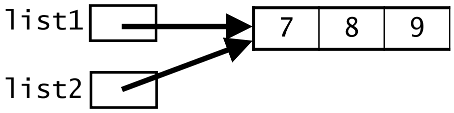


The variables are like two business cards that both have the address of the same office.

The list is the office.

- ...if we change *the internals* of the list...

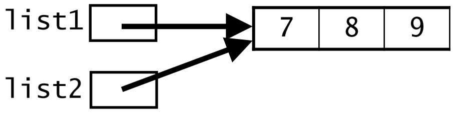
```
list2[2] = 4
print(list1)
```



Changing the Internals vs. Changing a Variable

- When two variables hold a reference to the same list...

```
list1 = [7, 8, 9]
list2 = list1
```

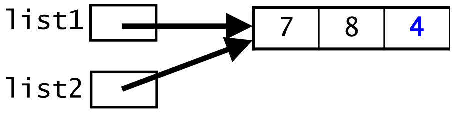


The variables are like two business cards that both have the address of the same office.

The list is the office.

- ...if we change *the internals* of the list, both variables will "see" the change:

```
list2[2] = 4
print(list1)     # prints [7, 8, 4]
```



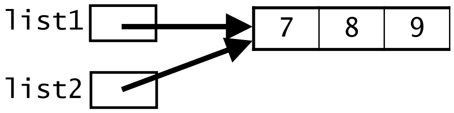
We're changing the contents of the office.

Using either business card to find the office will lead you to see the changed contents.

Changing the Internals vs. Changing a Variable (cont.)

- When two variables hold a reference to the same list...

```
list1 = [7, 8, 9]
list2 = list1
```

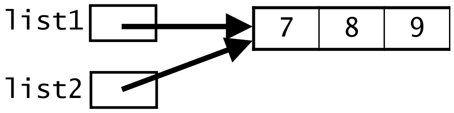


The variables are like two business cards that both have the address of the same office.

The list is the office.

- ...if we change one of the variables *itself*...

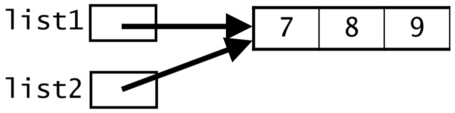
```
list2 = [4, 5, 6]
print(list1)
```



Changing the Internals vs. Changing a Variable (cont.)

- When two variables hold a reference to the same list...

```
list1 = [7, 8, 9]
list2 = list1
```

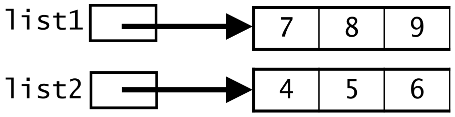


The variables are like two business cards that both have the address of the same office.

The list is the office.

- ...if we change one of the variables *itself*, that does *not* change the other variable:

```
list2 = [4, 5, 6]
print(list1)    # prints [7, 8, 9]
```



We're changing the address on one of the business cards.

It now refers to a different office.

The other business card still refers to the original unchanged office!

What is the output of this program?

```
def mystery5(x):
    x = x * -1
    return x
def mystery6(l1, l2):
    l1[0] = 0
    l2 = [1, 1]

x = 7
vals = [7, 7]
mystery5(x)
mystery6(vals, vals)
print(x, vals)
```

A. `7 [7, 7]`

B. `-7 [1, 1]`

C. `7 [0, 7]`

D. `7 [1, 1]`

E. `-7 [0, 7]`

What is the output of this program?

```
def mystery5(x):
    x = x * -1
    return x
def mystery6(l1, l2):
    l1[0] = 0
    l2 = [1, 1]

x = 7
vals = [7, 7]
mystery5(x)
mystery6(vals, vals)
print(x, vals)
```

A. `7 [7, 7]`

B. `-7 [1, 1]`

C. **`7 [0, 7]`**

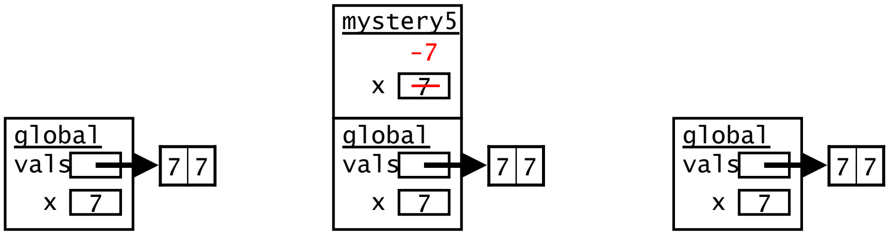
D. `7 [1, 1]`

E. `-7 [0, 7]`

What is the output of this program?

```
def mystery5(x):
    x = x * -1
    return x
def mystery6(l1, l2):
    l1[0] = 0
    l2 = [1, 1]

x = 7
vals = [7, 7]
mystery5(x)     # throw return value away!
mystery6(vals, vals)
print(x, vals)
```

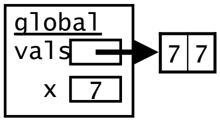
before mystery5 | *during* mystery5 | *after* mystery5



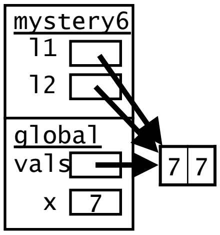
What is the output of this program?

```
def mystery5(x):
    x = x * -1
    return x
def mystery6(l1, l2):
    l1[0] = 0
    l2 = [1, 1]

x = 7
vals = [7, 7]
mystery5(x)
mystery6(vals, vals)
print(x, vals)
```

before mystery6



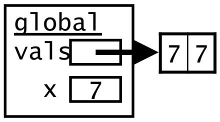
during mystery6



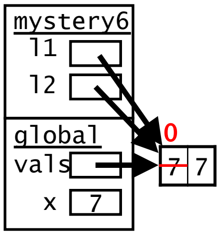
What is the output of this program?

```
def mystery5(x):
    x = x * -1
    return x
def mystery6(l1, l2):
    l1[0] = 0
    l2 = [1, 1]

x = 7
vals = [7, 7]
mystery5(x)
mystery6(vals, vals)
print(x, vals)
```

before mystery6



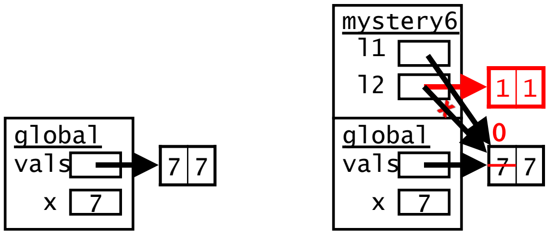
during mystery6



What is the output of this program?

```
def mystery5(x):
    x = x * -1
    return x
def mystery6(l1, l2):
    l1[0] = 0
    l2 = [1, 1]

x = 7
vals = [7, 7]
mystery5(x)
mystery6(vals, vals)
print(x, vals)
```

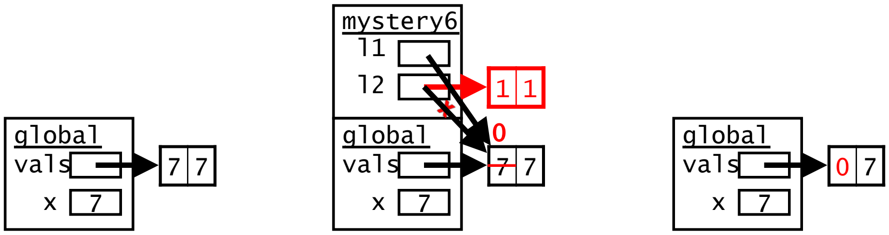
before mystery6 *during* mystery6



What is the output of this program?

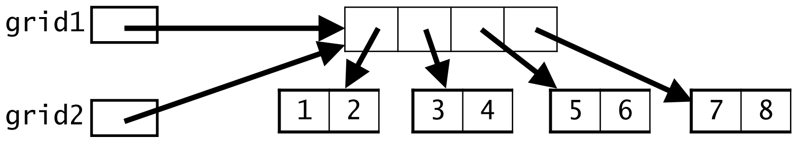
```
def mystery5(x):
    x = x * -1
    return x
def mystery6(l1, l2):
    l1[0] = 0
    l2 = [1, 1]

x = 7
vals = [7, 7]
mystery5(x)
mystery6(vals, vals)
print(x, vals)          # output: 7 [0, 7]
```

before mystery6 *during* mystery6 *after* mystery6



Copying a 2-D List

```
grid1 = [[1, 2], [3, 4], [5, 6], [7, 8]]
```



grid1

grid2

1	2

3	4

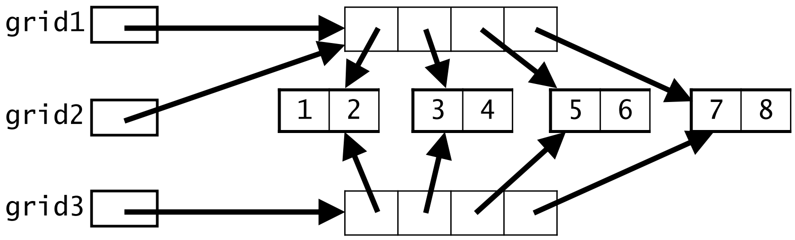
5	6

7	8

- This still doesn't copy the list: `grid2 = grid1`
- Does this? `grid3 = grid1[:]`

Copying a 2-D List

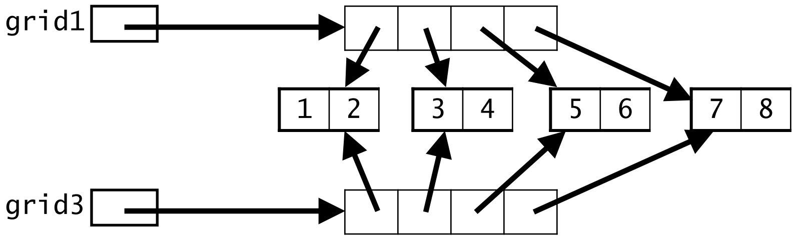
```
grid1 = [[1, 2], [3, 4], [5, 6], [7, 8]]
```



- This still doesn't copy the list: `grid2 = grid1`
- Does this? `grid3 = grid1`**`[:]`** **not fully!**

A *Shallow* Copy

```
grid1 = [[1, 2], [3, 4], [5, 6], [7, 8]]
grid3 = grid1[:]
```

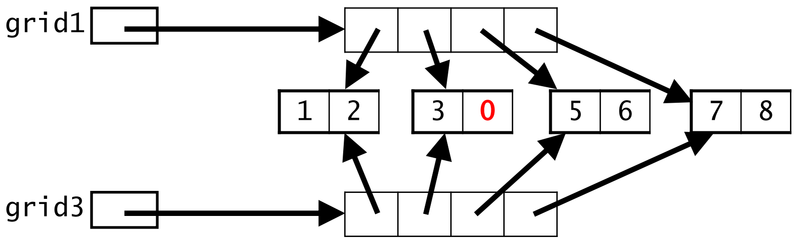


- `grid1` and `grid3` now share the same sublists.
  - known as a *shallow* copy
- What would this print?

```
grid1[1][1] = 0
print(grid3)
```

A *Shallow* Copy

```
grid1 = [[1, 2], [3, 4], [5, 6], [7, 8]]
grid3 = grid1[:]
```

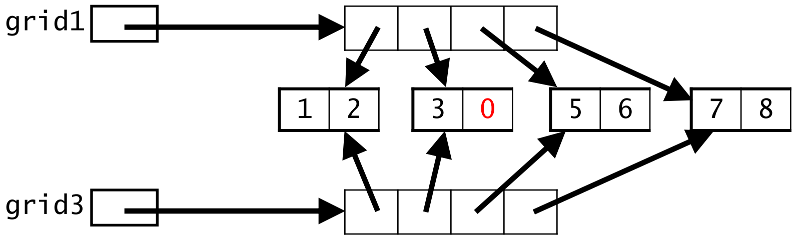


- `grid1 and grid3` now share the same sublists.
  - known as a *shallow* copy
- What would this print?

```
grid1[1][1] = 0
print(grid3)
```

A *Shallow* Copy

```
grid1 = [[1, 2], [3, 4], [5, 6], [7, 8]]
grid3 = grid1[:]
```

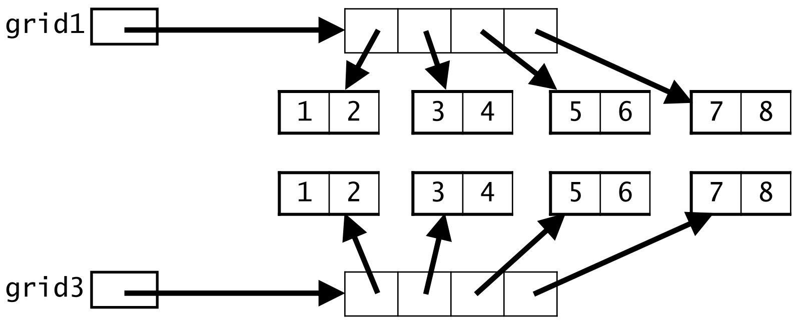


- `grid1 and grid3` now share the same sublists.
  - known as a *shallow* copy
- What would this print?

```
grid1[1][1] = 0
print(grid3)      [[1, 2], [3, 0], [5, 6], [7, 8]]
```

A *Deep* Copy: Nothing is Shared

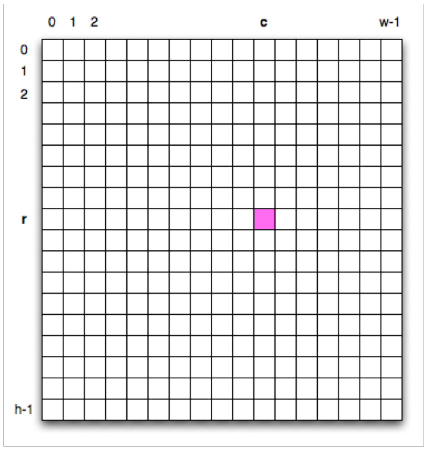
```
grid1 = [[1, 2], [3, 4], [5, 6], [7, 8]]
```



- In PS 7, you'll see one way to do this.

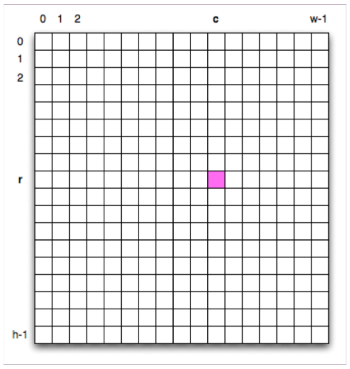
PS 7: Image Processing

- An image is a 2-D collection of *pixels*.
  - h rows, w columns
- The pixel at position (r, c) tells you the color of the image at that location.
- We'll load an image's pixels into a 2-D list and process it:



```
pixels = load_pixels('my_image.png') #get a 2-D list!
h = len(pixels)
w = len(pixels[0])
for r in range(h):
    for c in range(w):
        # process pixels[r, c] in some way
```

Pixels in PS 7



- Each pixel is represented by a list of 3 integers that specify its color:

  `[red, green, blue]`

  - example: the pink pixel at right has color

    `[240, 60, 225]`

  - known as RGB values
  - each value is between 0-255

- Other examples:
  - pure red: `[255, 0, 0]`
  - pure green: `[0, 255, 0]`
  - pure blue: `[0, 0, 255]`
  - white: `[255, 255, 255]`
  - black: `[0, 0, 0]`