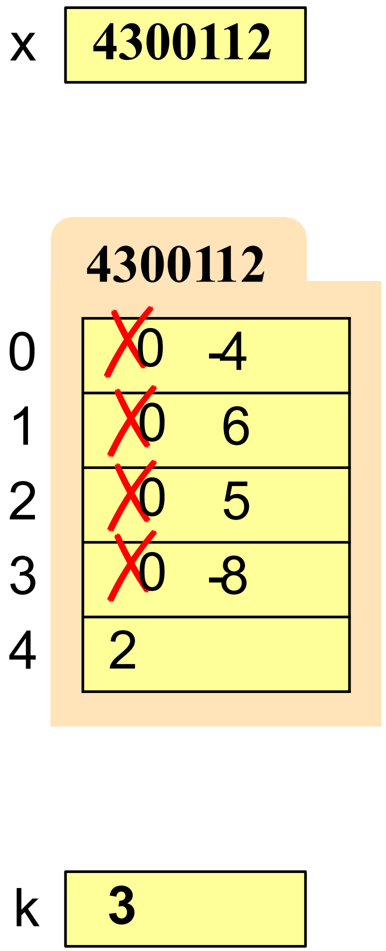
Lists and Sequences

Overview of List Syntax

- x = [0, 0, 0, 0]	Create list of length 4 with all zeroes
- x.append(2)	Append 2 to end of list x (now length 5)
- 3 in x	Evaluates to False (3 not in x)
- x[2] = 5 - x[0] = –4	Assign 5 to element 2 and –4 to element 0
- k = 3 - x[k] = 2 * x[0] - x[k–2] = 6	Assign -8to x[3] and 6 to x[1]



Lists vs. Tuples vs. Strings

- **Creation**

  x = [a1, a2, a3, …]

  Can contain anything
- **len(x) is length**
- **Supports slicing**

  Example: x[1:2]

  x[i] is an element
- **Can concatenate** y = x + [1, 2] Makes a new list
- **Is mutable**

  x.append(5)

- **Creation**

  x = (a1, a2, a3, …)

  Can contain anything
- **len(x) is length**
- **Supports slicing**

  Example: x[1:2]

  x[i] is an element
- **Can concatenate** y = x + (1, 2) Makes a new tuple
- **Is not mutable**

- **Creation**

  x = 'Hello'

  Only contains chars
- **len(x) is length**
- **Supports slicing**

  Example: x[1:2]

  x[i] is a substring
- **Can concatenate**

  y = x + ' World'

  Makes a new string
- **Is not mutable**

Lists vs. Tuples vs. Strings

- **Creation**

  x = [a1, a2, a3, …]

  Can contain anything
- **len(x) is length**
- **Supports slicing**

  Example: x[1:2]

  x[i] is an element
- **Can concatenate** y = x + [1, 2] Makes a new list
- **Is mutable**

  x.append(5)

- **Creation**

  x = (a1, a2, a

  Can contain
- **len(x) is length**
- **Supports slicing**

  Example: x[1:2]

  x[i] is an element
- **Can concatenate** y = x + (1, 2) Makes a new tuple
- **Is not mutable**

Did not use this semester but works almost like lists do

ars

- **len(x) is length**
- **Supports slicing**

  Example: x[1:2]

  x[i] is a substring
- **Can concatenate**

  y = x + ' World'

  Makes a new string
- **Is not mutable**

Quick for loop review

Basic Structure:

```
for <placeholder variable> in <list to loop through>:
    do something...
```

Two general forms:

```
thelist = [’a’, ‘b’, ‘c’, ‘d’]
for foo in thelist:
    print foo
```

Loops through the elements of thelist

```
thelist = [’a’, ‘b’, ‘c’, ‘d’]
for index in range(len(thelist)):
    print thelist[index]
```

Loops through the indicies of thelist

Think about what range really returns!

```
range(4)  >>  [0,1,2,3]
range(1)  >>  [0]
```

Modified Question 4 from Fall 2011

Each elements in the list scores contains the number of students who received score i on a test. For example, if 30 students got 85, then scores[85] is 30.Write the body of function histogram, which returns a histogram as a list of strings. (You need not write loop invariants.) For example, if scores = [7, 0, 4, 3, 2, 0, …] then the first elements of the resulting string list are:

```
'00 *******'
'01 '
'02 ****'
'03 ***'
'04 *'
'05 '
```

Modified Question 4 from Fall 2011

```
def histogram(scores):
    """Return a list of Strings (call it s) in which each s[i] contains:
        (1) i, as  a  two-digit integer (with leading zeros if necessary)
        (2) a blank,
        (3) n asterisks '*', where  n is scores[i].
    Precondition: scores is a list of nonnegative integers,
    len(scores) < 100"""
    # IMPLEMENT ME
```

Modified Question 4 from Fall 2011

```
def histogram(scores):
    """Return a list of Strings (call it s) in which each s[i] contains:
        (1)  i, as  a  two-digit integer (with leading zeros if necessary)
        (2)  a blank,
        (3)  n asterisks '*', where  n is scores[i].
    Precondition: scores is a list of nonnegative integers, len(scores) < 100"""
    s = []    # List to contain the result.
    for i in range(len(scores)):        # Need the value i, not the elements of scores
        if scores[i] < 10:
            row = str(scores[i]) + ‘ ‘
        else:
            row = ‘0’ + str(scores[i]) + ‘ ‘     # Add a 0 for double digits
        for n in range(scores[i]):
            row = row + ‘*’                        # Append scores[i] number of asterisks
        s.append(row)
    return s
```

Overview of Two-Dimensional Lists

- Access value at row 3, col 2:

  d[3][2]

- Assign value at row 3, col 2:

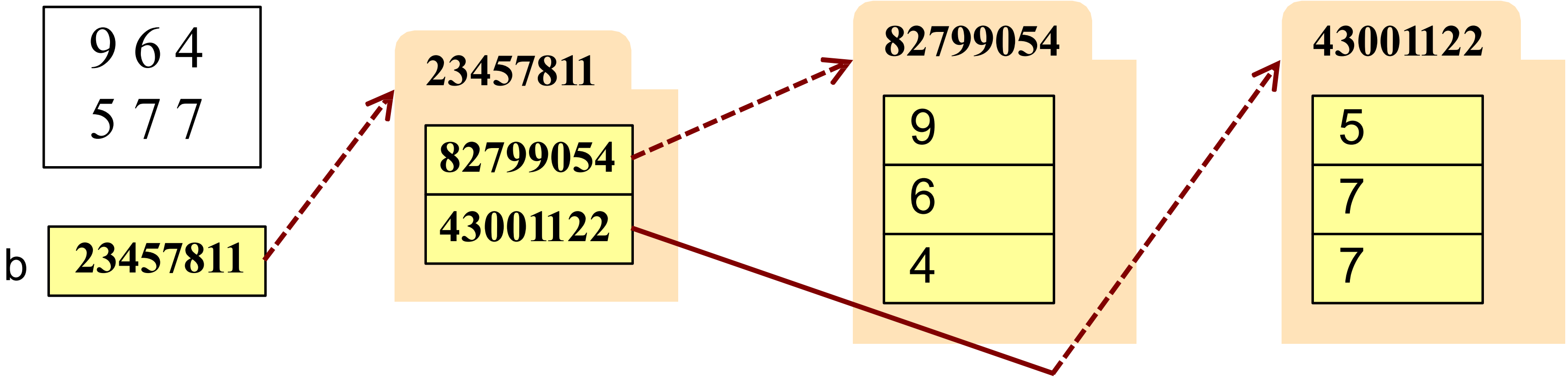
  d[3][2] = 8

- **An odd symmetry**
  - Number of rows of d: len(d)
  - Number of cols in row r of d: len(d[r])

d	0	1	2	3
0	5	4	7	3
1	4	8	9	7
2	5	1	2	3
3	4	1	2	9
4	6	7	8	0

How Multidimensional Lists are Stored

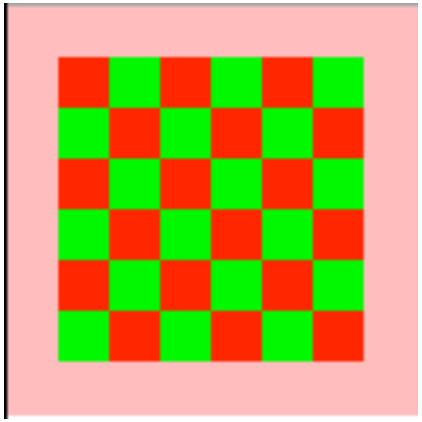
- b = [[9, 6, 4], [5, 7, 7]]



- b holds name of a one-dimensional list
  - Has len(b) elements
  - Its elements are (the names of) 1D lists
- b[i] holds the name of a one-dimensional list (of ints)
  - Has len(b[i]) elements

Modified Question 4 from Fall 2010

Recall drawing GRectangles in A7. Write method placeSquares, whose requirements appear below. It draws square bricks as shown to the right and returns them as a 2d list of GRectangle



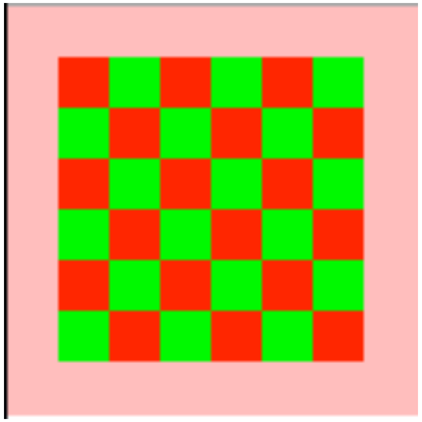
```
def placeSquares(self, m):
    """Create a list of m x m squares (GRectangle), as
    specified below, adding the squares to the GUI, and
    return the list."""
```

Method Requirements:

- There are m columns and rows of squares; precondition: 0 < m.
- Each square has side length BRICK_SIDE; there is no space between them.
- The bottom-left square is at the bottom-left corner (0,0) of the GUI. Squares in columns and rows 0 and m-1 have color colormodel.PINK
- Inner squares have checkerboard pattern of colormodel.RED and colormodel.GREEN, as shown (bottom-left one is green; one next to it, red).

Modified Question 4 from Fall 2010

Recall drawing GRectangles in A7. Write method placeSquares, whose requirements appear below. It draws square bricks as shown to the right and returns them as a 2d list of GRectangle



```
def placeSquares(self, m):
    """Create a list of m x m squares (GRectangle), as
    specified on last slide, adding them to the GUI, and
    return the list."""
```

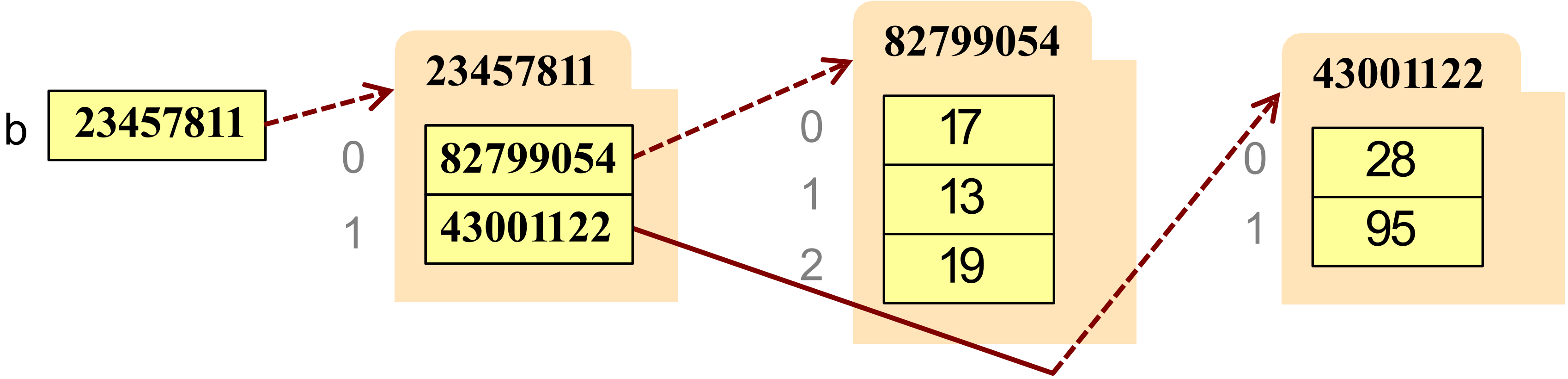
API Reminders:

- GRectangle has attributes pos (a 2 element tuple), size (a 2 element tuple), fillcolor, and linecolor
- You construct a GRectangle with keyword arguments: GRectangle(pos=(0,0),size=(10,10))
- You add to the GUI with self.view.add(…)

```
def  placeSquares(self, m):
    """Place the m x m Bricks, as requested on the exam and return the list"""
    bricks = [];  r  = 0           # Make a new list to represent the whole grid
    while r < m:                   # Place col c of bricks
        row = [];  c = 0           # Make  a new  list to represent rows
        while c < m:
            color = colormodel.RED
            if r == 0 or r == m-1 or c == 0 or c == m-1:
                color = colormodel.PINK
            elif r+c % 2 ==  0:
                color = colormodel.GREEN
            brick=GRectangle(pos=(r*BRICK_SIDE,c*BRICK_SIDE), fillcolor=color
                             size=(BRICK_SIDE,BRICK_SIDE), linecolor=color)
            row.append(brick)
            self.view.add(brick)
            c = c+1
        bricks.append(row)
        r = r+1
    return bricks
```

Ragged Lists: Rows w/ Different Length

- b = [[17,13,19],[28,95]]



- To create a ragged list
  - Create b as an empty list (b = [])
  - Create each row as a list (r1 = [17,13,19]; r2 = [28,95])
  - Append lists to b (b.append(r1); b.append(r2))

Modified Question 4 from Fall 2011

Someone messed up a method to create certain arrays for us. For example (and this is only an example), they produced the array:

```
3 1 2                                 1 2 3
2 1 7 8 5        instead of           1 7 8 5 2
5                the array            5
6 8                                   8 6
```

Thus, they put the last value of each row at the beginning instead of the end.

Write a procedure that fixes this by rotating each row one position to the left; each element is moved one position earlier, and the first element is placed in the last position. Do not use recursion. **DO NOT RETURN A VALUE**.

```
def rotate(b):
   """Rotate each row one position to the left, as explained
   above.  Precondition: b  is a  list, might be  ragged,  and each
   row has >= 1 value"""
```

Modified Question 4 from Fall 2011

```
def rotate(b):
    """Rotate each row one position to the left, as explained on the previous
    slide.  Precondition:  b is a list, might be  ragged, and each  row has >= 1
    value"""
    # invariant: rows 0..r–1 of b have been rotated
    r = 0
    while r < len(b):
        first = b[r][0]                # Rotate row r one position to the left;
        # inv: b[r][1..c–1] moved to b[r][0..c–2]
        c = 1
        while c < len(b[r])
            b[r][c-1]= b[r][c];
            c= c+1
        # post: b[r][1..] has been  moved  to b[r][0..]
        b[r][len(b[r])–1]= first
        r = r+1
    # post: rows 0..b.length–1 of b  has been  rotated
```

Dictionaries

Overview of Dictionary Syntax

- **Creation**

```
d = dict()
d = {}
```

These two do the exact same thing! Creates an empty dictionary

- **Insertion**

```
d[’new_key’] = ‘new_value’
```

Adds ’new_value’ to d with the key of ‘new_key’

- **Modification**

```
d[’new_key’] = ‘even_newer_value’
```

Changes the value at ‘new_key’ to ‘even_newer_value’

Note: Insertion and Modification has the same syntax! Whether it modifies or not depends on if the key is already in the dictionary

Overview of Dictionary Syntax

- **Creation**

  d = dict()
  d = {}

These two do the exact same thing! Creates an empty dictionary

- **Insertion**

  d['new_key'] = 'new_value'

Adds 'new_value' to d with the key of 'new_key'

- **Modification**

  d['new_key'] = 'even_newer_value'

Changes the value at 'new_key' to 'even_newer_value'

- **Search**

  'new_key' in d >> returns **True**
  'random_key' in d >> returns **False**

Use the 'in' keyword to check if a key is in the dictionary

- **Deletion**

  del d['new_key']

Deletes key-value pair: 'new_key' is removed along with its value, 'even_newer_value'

Histograms Revisited (Dictionaries)

```
def histogram(scores):
    """Return a histogram where the key value pair is:
            (score, number of occurrences)
       so that every score in scores is represented.
       If there a score is not in scores, then it does not need to be
       reflected in the dictionary with (score, 0).
    Precondition: scores is a list of nonnegative integers, len(scores) <
    100"""
    # IMPLEMENT ME
```

Histograms Revisited (Dictionaries)

```
def histogram(scores):
    """Return a histogram where the key value pair is:
             (score, number of occurrences)
        so that every score in scores is represented.
        If there a score is not in scores, then it does not need to be
        reflected in the dictionary with (score, 0).
    Precondition: scores is a list of nonnegative integers, ‘
    len(scores) < 100""
    histogram = dict()     # Could have also written histogram = {}
    for score in scores:
        if score in histogram:   # Check if this score is already in histogram
            histogram[score] += 1
        else:
            histogram[score] = 1
    return histogram
```

Histograms Revisited (Dictionaries)

```
def histogram(scores):
    """Return a histogram where the key value pair is:
            (score, number of occurrences)
       so that every score in scores is represented.
       If there a score is not in scores, then it does not need to be
       reflected in the dictionary with (score, 0).
    Precondition: scores is a list of nonnegative integers, ‘
    len(scores) < 100""
    histogram = dict()     # Could have also written histogram = {}
    for score in scores:
       if score in histogram:   # Check if this score is already in histogram
           histogram[score] += 1
       else:
           histogram[score] = 1
    return histogram
```

Very common idiom. Make sure you’re familiar with it!

Python Basics

Basic Types

- Strings (str)

  Literals surrounded in quotes: “Hello World!”

- Booleans (bool)

  Two possible values: **True** or **False**

- Integers (int)

  Represents whole numbers: …-1, 0, 1, 2, 3…

- Floats (float)

  Represents decimals: -0.1, 1.4445, 2.48935,…

Booleans (bool)

Represents **logical statements!**

Operators: not, and, or

- not b: **True** if b is false and **False** if b is true (negation)
- a and b: **True** if **both** a and b are true and **False** otherwise.
- a or b: **True** if a is true **or** b is true and **False** otherwise.

Often are results of comparisons:

- Order comparison:
  - a < b; a <=b; a >= b; a > b
- Equality comparison:
  - a == b; a != b

Short Circuiting:

- (False and x / 0) vs (x / 0 and False)
- (True or x / 0) vs (x / 0 or True)

Strings (str)

Used to represent **text**.

Anything surrounded in either single quotes or double quotes is a string.

Operators: + (concatenation)

- “Hello ” + ‘World!’ >> “Hello World!”

Don’t forget about string **methods!** A few common ones:

- find() and index(); know the difference and what the second optional argument does
- count()
- split()
- join()

String indexing and splicing:

- You access specific indexes using s[i] where s is the **str** and i is an **int**
- Splice substrings using s[i:j]. i is **inclusive** while j is **exclusive**

If-statements

Basic Structure:

```
if <boolean expression>:
    do something…
else:
    do something…
```

This lets you control the **flow** of your code, directing it down branches depending on certain variables!

Common style problem:

```
if x == True:     # Think about what the type of x is!
    do something…
else:
    do something…
```

Questions