Module 18

Nested Lists

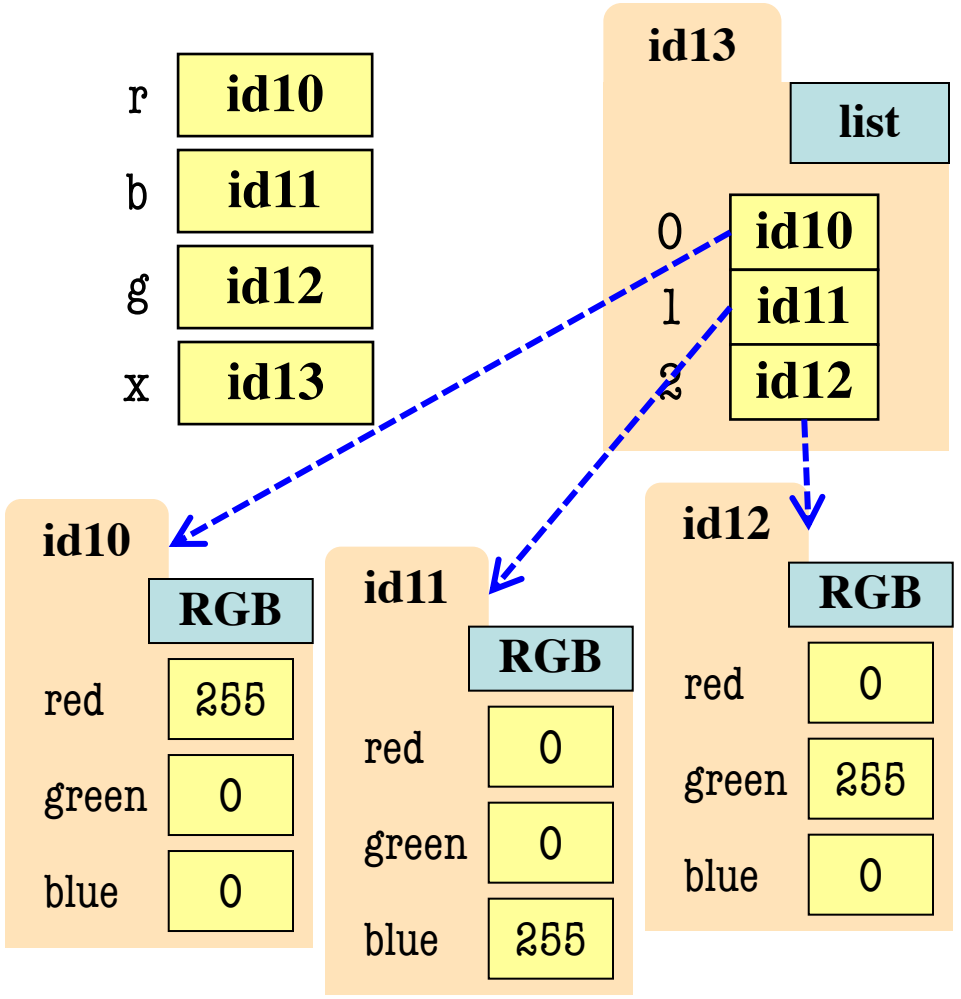
Motivating this Course Module

- In previous video series, introduced lists
  - All our examples had *primitives* in the lists
  - So we only showed one folder in visualization
  - **Demo:** `x = [5,6,5,9]`
- But lists can contain anything!
  - What if they can contain objects?
  - Yes, primitives = objects, but ignore folders
  - We mean if we cannot ignore the folders

Lists of Objects

- List positions are variables
  - Can store base types
  - But cannot store folders
  - Can store folder identifiers
- Folders linking to folders
  - Top folder for the list
  - Other folders for contents
- Example:

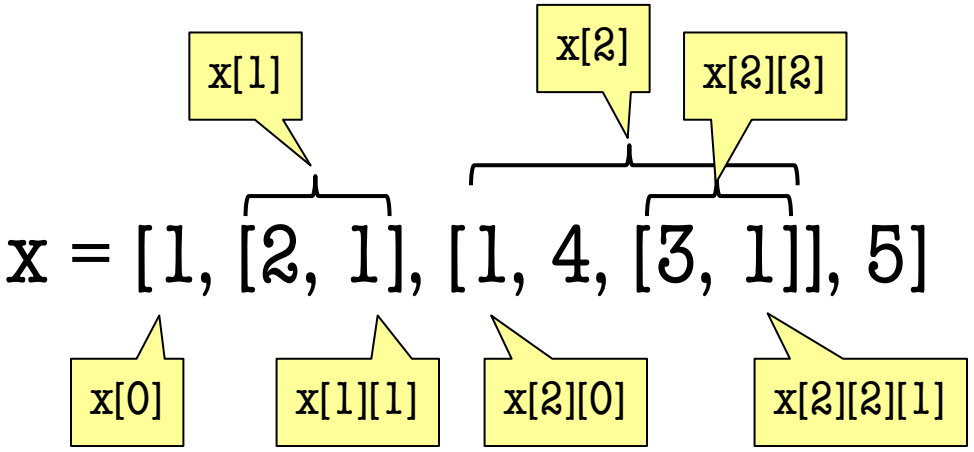
```
>>> r = introcs.RGB(255,0,0)
>>> b = introcs.RGB(0,0,255)
>>> g = introcs.RGB(0,255,0)
>>> x = [r,b,g]
```



Nested Lists

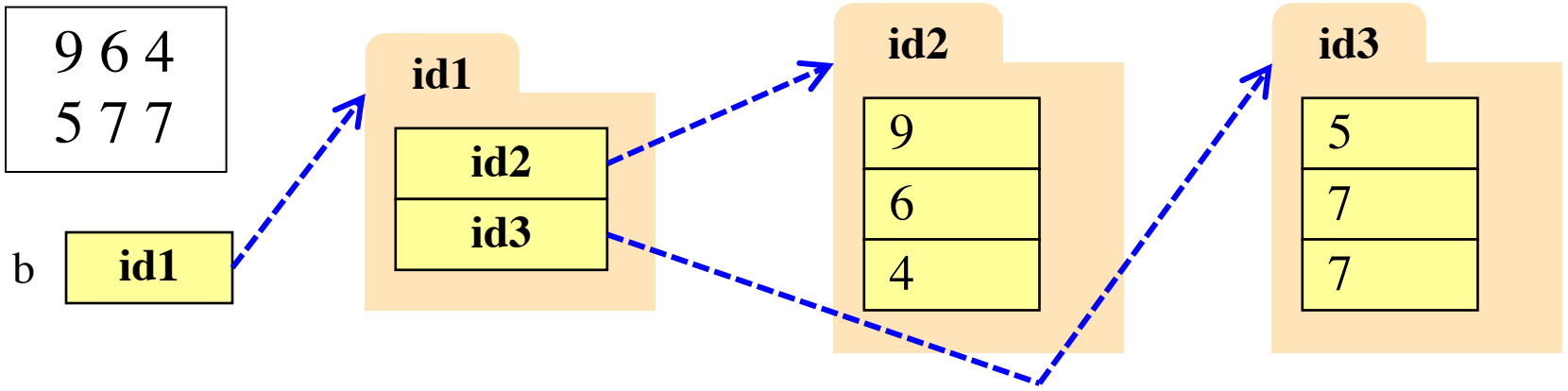
- We have seen that lists can hold any objects
  - Lists are objects
  - Therefore lists can hold other lists!
- Known as **nested or multidimensional list**

```
a = [2, 1]
b = [3, 1]
c = [1, 4, b]
x = [1, a, c, 5]
```



How Multidimensional Lists are Stored

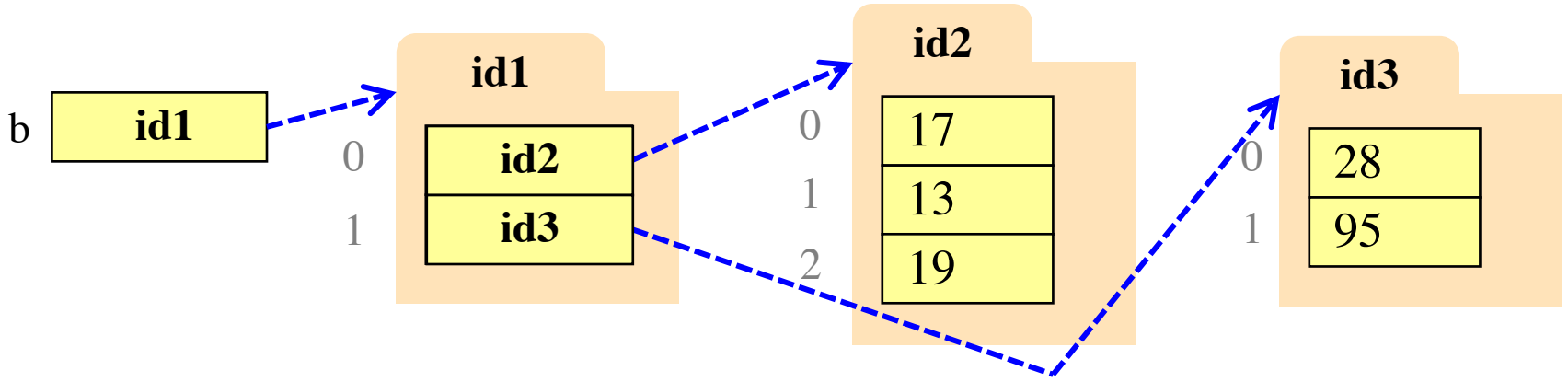
- b = [[9, 6, 4], [5, 7, 7]]



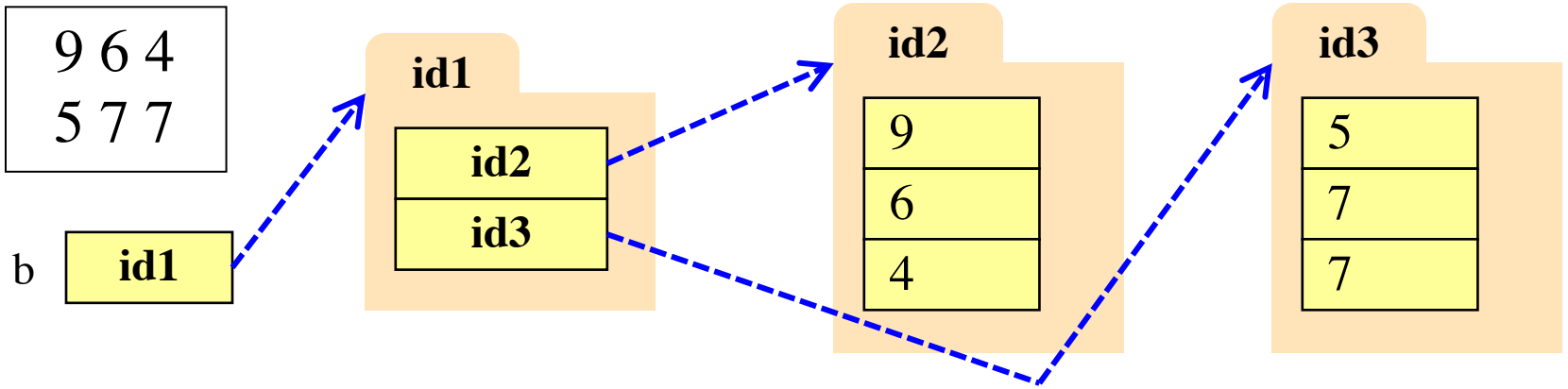
- b holds name of a one-dimensional list
  - Has len(b) elements
  - Its elements are (the names of) 1D lists
- b[i] holds the name of a one-dimensional list (of ints)
  - Has len(b[i]) elements

Ragged Lists vs Tables

- Ragged is 2d uneven list: b = [[17,13,19],[28,95]]



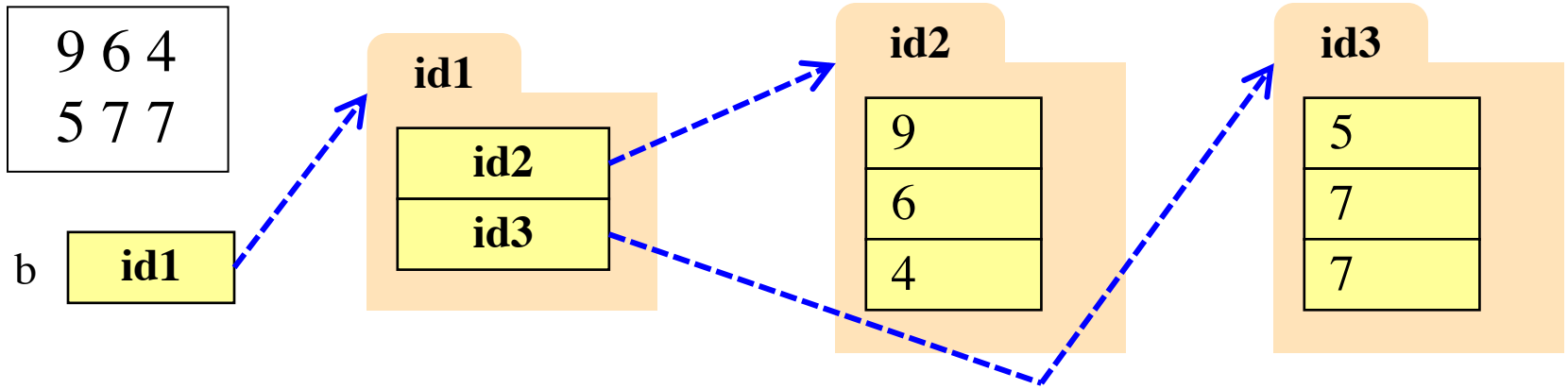
- Table is 2d uniform list: b = [[9,6,4],[5,7,7]]



Ragged Lists vs Tables

- Ragged is 2d uneven list: `b = [[17,13,19],[28,95]]`

- Table is 2d uniform list: `b = [[9,6,4],[5,7,7]]`



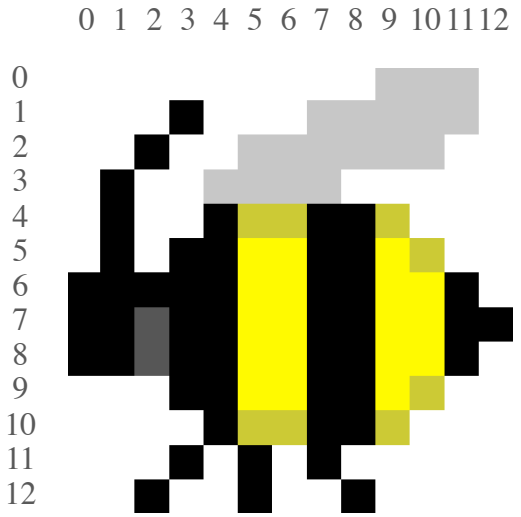
Tables are a Common Type of Data

Spreadsheet

	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

table.csv

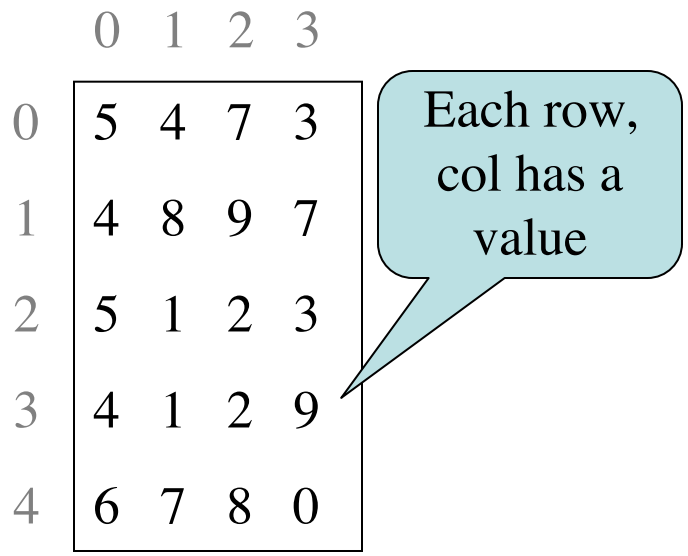
Image



bee.xlsx

Representing Tables as Lists

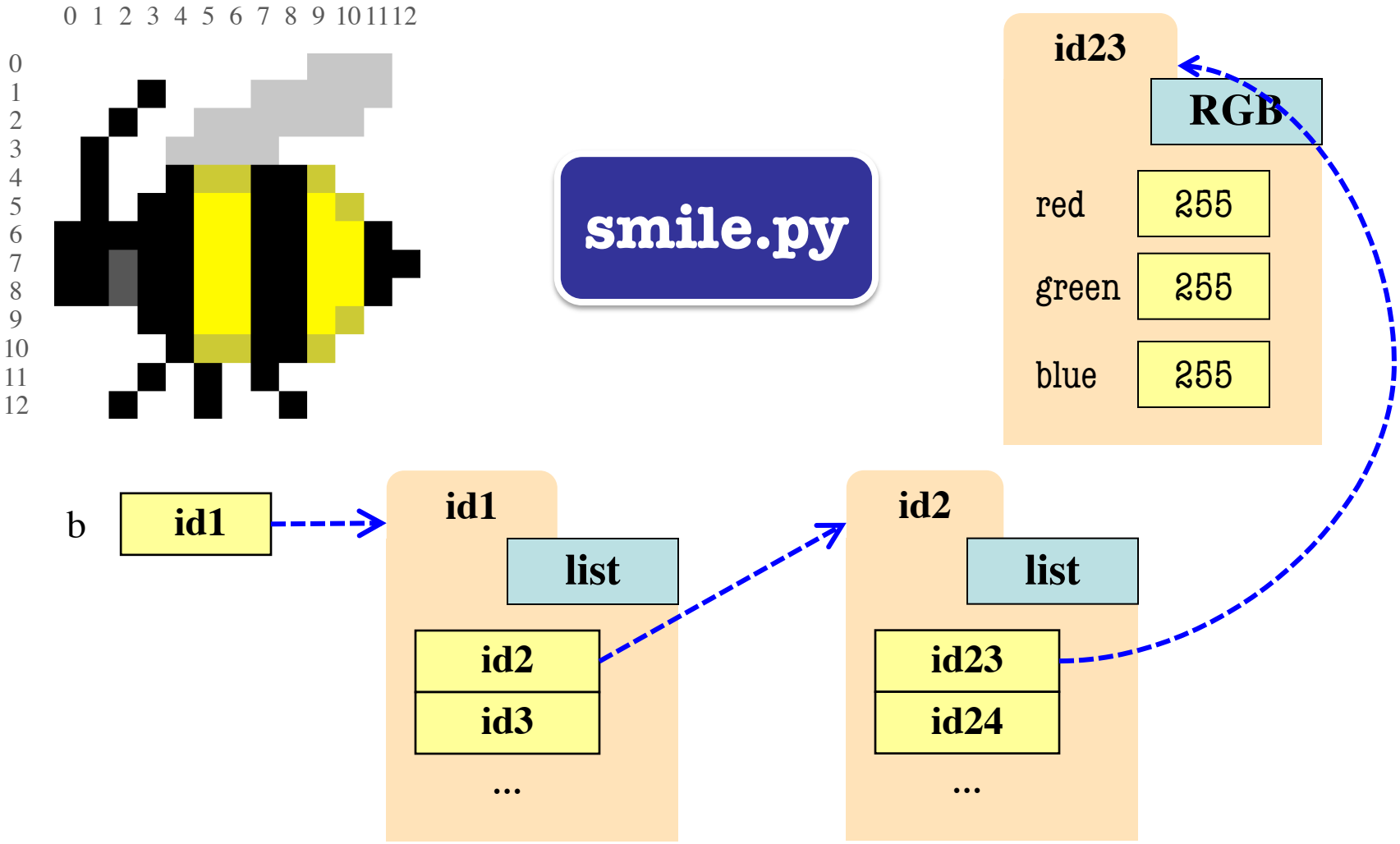
Spreadsheet



- Represent as 2d list
  - Each table row a list
  - List of all rows
  - **Row major order**
- Column major exists
  - Less common to see
  - Limited to some scientific applications

```
d = [[5,4,7,3],[4,8,9,7],[5,1,2,3],[4,1,2,9],[6,7,8,0]]
```

Image Data: 2D Lists of Pixels



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`

- **There is an odd symmetry**
  - Number of rows of d: `len(d)`
  - Number of cols in row r of d: `len(d[r])`

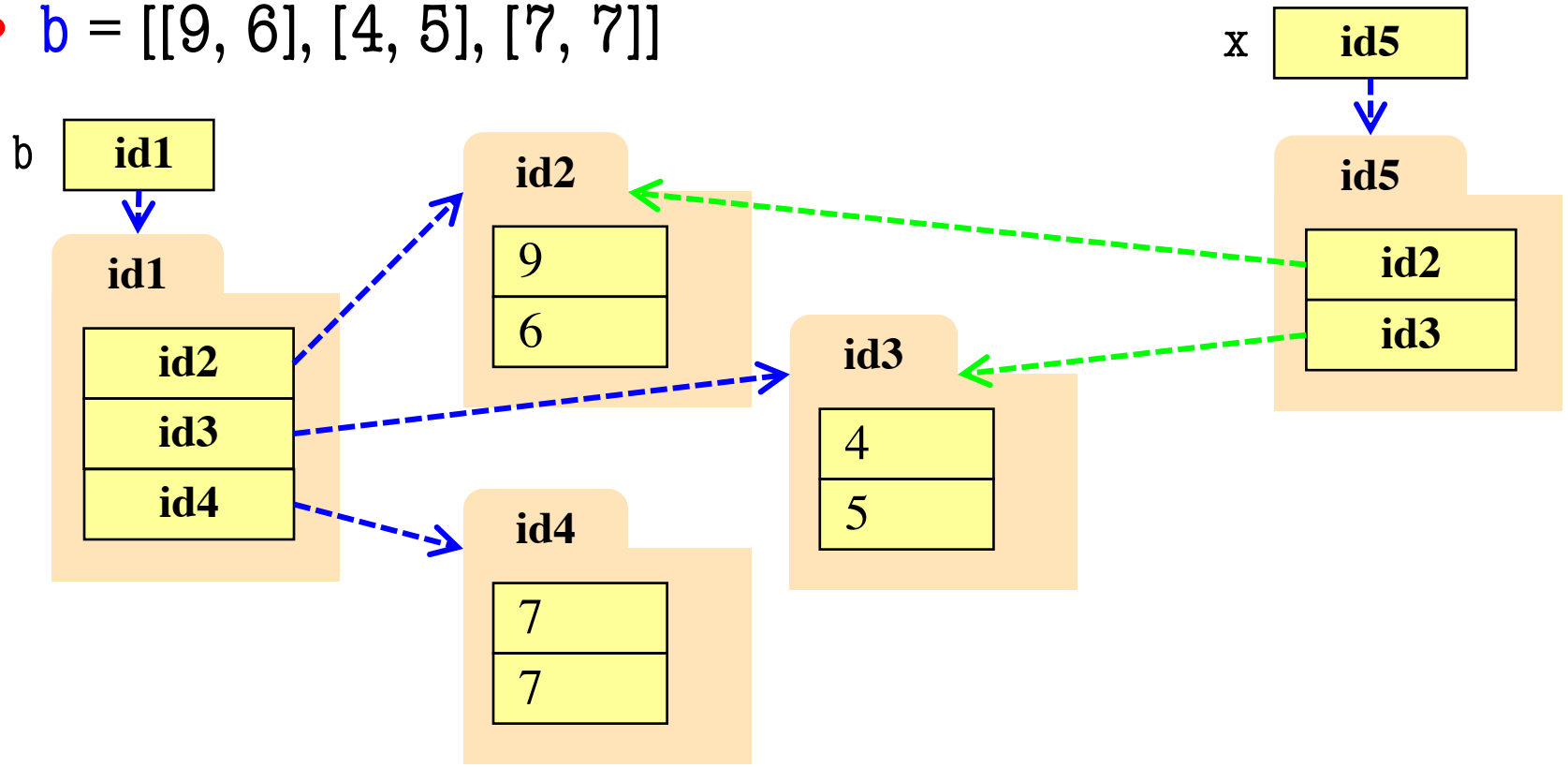
`d`

	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

Slices and Multidimensional Lists

- Only “top-level” list is copied.
- Contents of the list are not altered
- `b` = [[9, 6], [4, 5], [7, 7]]

`x = b[:2]`



Slices and Multidimensional Lists

- Create a nested list

```
>>> b = [[9,6],[4,5],[7,7]]
```

- Get a slice

```
>>> x = b[:2]
```

- Append to a row of x

```
>>> x[1].append(10)
```

- x now has nested list

```
[[9, 6], [4, 5, 10]]
```

- What are the contents of the list (with name) in b?

A: [[9,6],[4,5],[7,7]]
B: [[9,6],[4,5,10]]
C: [[9,6],[4,5,10],[7,7]]
D: [[9,6],[4,10],[7,7]]
E: I don't know

Shallow vs. Deep Copy

- **Shallow copy:** Copy top-level list
  - Happens when slice a multidimensional list
- **Deep copy:** Copy top and all nested lists
  - Requires a special function: `copy.deepcopy`
- **Example:**

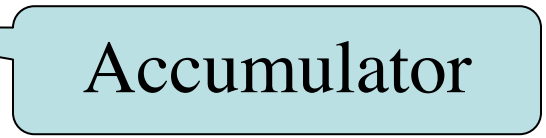
```
>>> import copy
>>> a = [[1,2],[2,3]]
>>> b = a[:]                    # Shallow copy
>>> c = copy.deepcopy(a)        # Deep copy
```

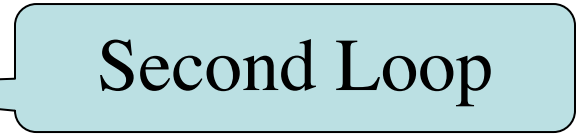
Relationship to Standard Lists

- Functions on nested lists similar to lists
  - Go over (nested) list with *for-loop*
  - Use *accumulator* to gather the results
- But two important differences
  - Need **multiple for-loops**
  - One for each part/dimension of loop
  - In some cases need **multiple accumulators**
  - Latter true when result is new table

Simple Example

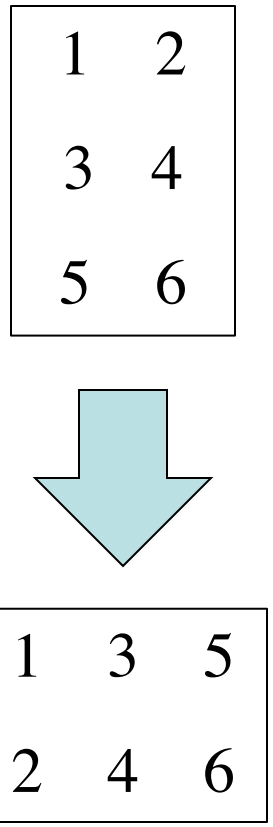
```
def all_nums(table):
    """Returns True if table contains only numbers

    Precondition: table is a (non-ragged) 2d List"""
    result = True
    # Walk through table
    for row in table:
        # Walk through the row
        for item in row:
            if not type(item) in [int,float]:
                result = False
    return result
```





Transpose: A Trickier Example

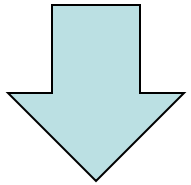
```
def transpose(table):
    """Returns: copy of table with rows and columns swapped

    Precondition: table is a (non-ragged) 2d List"""



    result = []                    # Result (new table) accumulator
    # Loop over columns
        # Add each column as a row to result




    return result
```



Transpose: A Trickier Example

```
def transpose(table):
    """Returns: copy of table with rows and columns swapped
    Precondition: table is a (non-ragged) 2d List"""
    numrows = len(table)     # Need number of rows
    numcols  = len(table[0]) # All rows have same no. cols
    result = []              # Result (new table) accumulator
    for m in range(numcols):
        # Get the column elements at position m
        # Make a new list for this column
        # Add this row to accumulator table

    return result
```

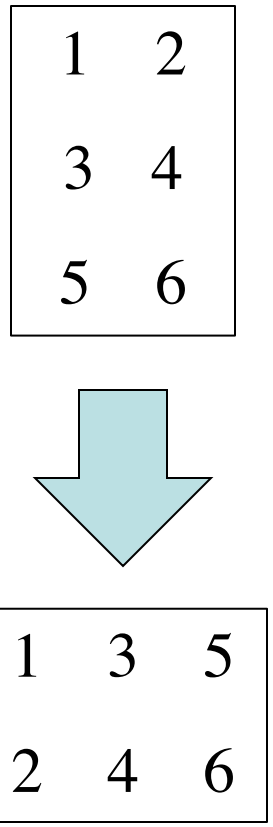
1	2
3	4
5	6



1	3	5
2	4	6

Transpose: A Trickier Example

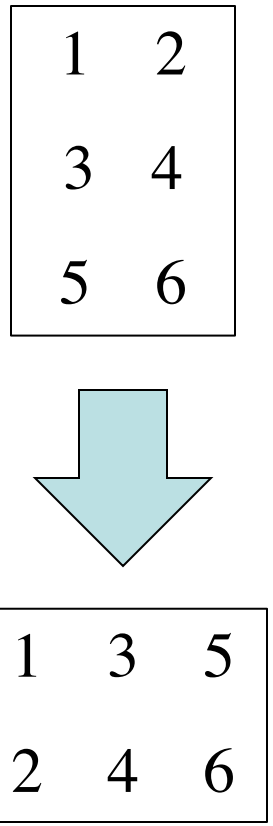
```
def transpose(table):
    """Returns: copy of table with rows and columns swapped
    Precondition: table is a (non-ragged) 2d List"""
    numrows = len(table)     # Need number of rows
    numcols  = len(table[0]) # All rows have same no. cols
    result = []              # Result (new table) accumulator
    for m in range(numcols):
        row = []                        # Single row accumulator
        for n in range(numrows):
            row.append(table[n][m]) # Create a new row list
        result.append(row)              # Add result to table
    return result
```



Transpose: A Trickier Example

```
def transpose(table):
    """Returns: copy of table with rows and columns swapped
    Precondition: table is a (non-ragged) 2d List"""
    numrows = len(table)     # Need number of rows
    numcols  = len(table[0]) # All rows have same no. cols
    result = []                                  ) accumulator
    for m in range(numc
        row = []                                 accumulator
        for n in range(numrows):
            row.append(table[n][m]) # Create a new row list
        result.append(row)          # Add result to table
    return result
```

Accumulator for each loop



Relationship to Standard Lists

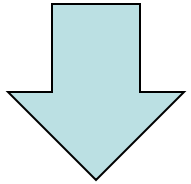
- Functions on nested lists similar to lists
  - Do not loop over the list (modifying it)
  - Loop over *range of positions* instead
  - No *accumulator* or return statement
- But one important difference
  - May need **multiple for-loops**
  - Depends on if modifying rows or entries

Simple Example

```
def add_ones(table):
    """Adds one to every number in the table
    Preconditions: table is a 2d List,
    all table elements are int"""
    # Walk through table
    for rpos in range(len(table)):
        # Walk through each column
        for cpos in range(len(table[rpos])):
            table[rpos][cpos] = table[rpos][cpos]+1

    # No return statement
```

Do not loop over the table

1	3	5
2	4	6



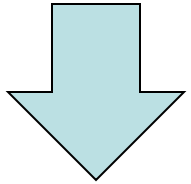
2	4	6
3	5	7

Simple Example

```
def add_ones(table):
    """Adds one to every number in the table
    Preconditions: table is a 2d List,
    all table elements are int"""
    # Walk through table
    for rpos in range(len(table)):
        # Walk through each column
        for cpos in range(len(table[rpos])):
            table[rpos][cpos] = table[rpos][cpos]+1

    # No return statement
```

Note that ragged is okay

1	3	5
2	4	6



2	4	6
3	5	7

Another Example

```
def strip(table,col):
    """Removes column col from the given table
    Preconditions: table is a (non-ragged) 2d List,
    col valid column"""
    # Walk through table
    for rpos in range(len(table)):
        # Modify each row to slice out column
        table[rpos] = table[rpos][:col] + table[rpos][col+1:]


    # No return statement
```

Do not loop over the table