Lecture 14

Nested Lists

Announcements for This Lecture

(Optional) Videos

- **Lesson 18** for today
- **Videos 17.1-17.5** next time

- **Prelim, 10/16 at 7:30 pm**
  - Material up to **10/2**
  - Study guide is posted
- **Conflicts are now closed!**
  - Only honoring emergencies
  - SDS students are notified

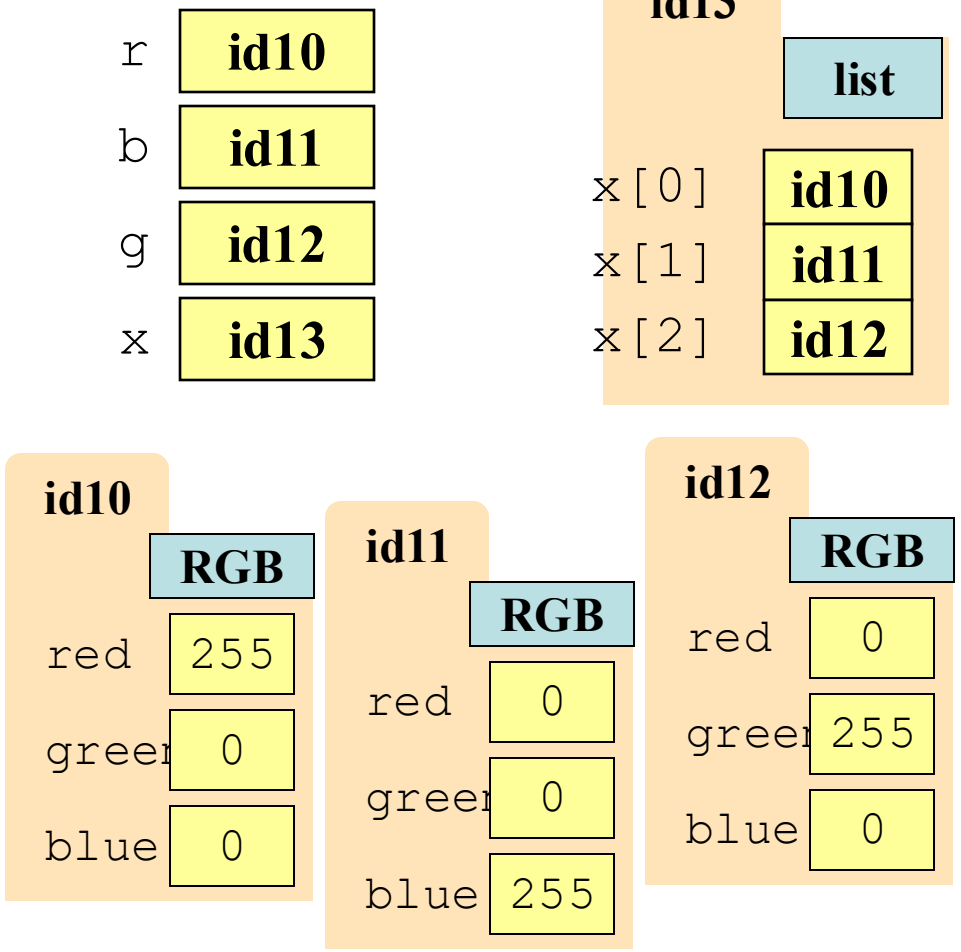
Assignments/Lab

- Labs this week are special
  - **Lab 13** due **after break**
  - **Lab 14** is **in-class only**
- A3 to be graded before exam
  - Survey is still active
- A4 after exam and break
  - Longer time to do this one
  - Covers for-loops
  - Also material next week

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:

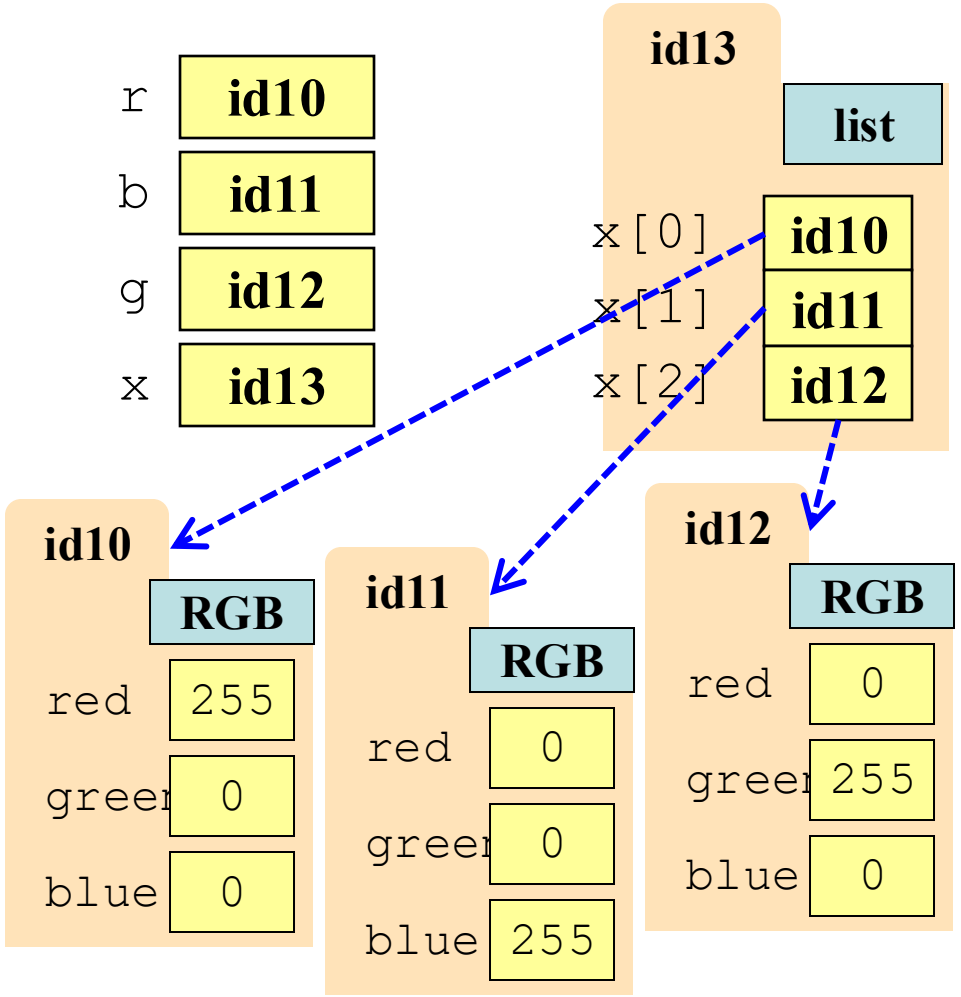
```
>>> r =
introcs.RGB(255,0,0)
>>> b =
introcs.RGB(0,0,255)
>>> g =
introcs.RGB(0,255,0)
>>> x = [r,b,g]
```



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:

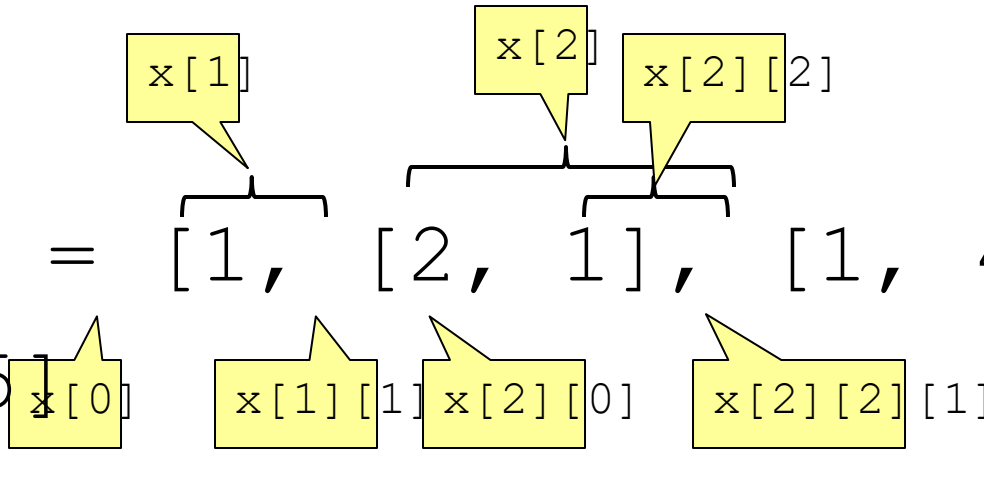
```
>>> r =
introcs.RGB(255,0,0)
>>> b =
introcs.RGB(0,0,255)
>>> g =
introcs.RGB(0,255,0)
>>> x = [r,b,g]
```



Nested Lists

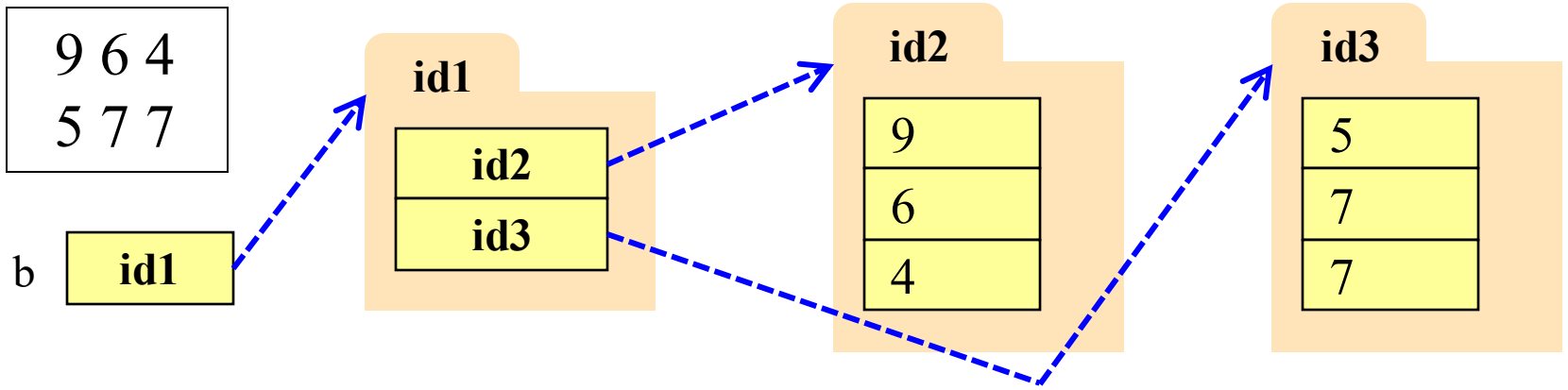
- Lists can hold any objects
- Lists are objects
- Therefore lists can hold other lists!

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



How Multidimensional Lists are Stored

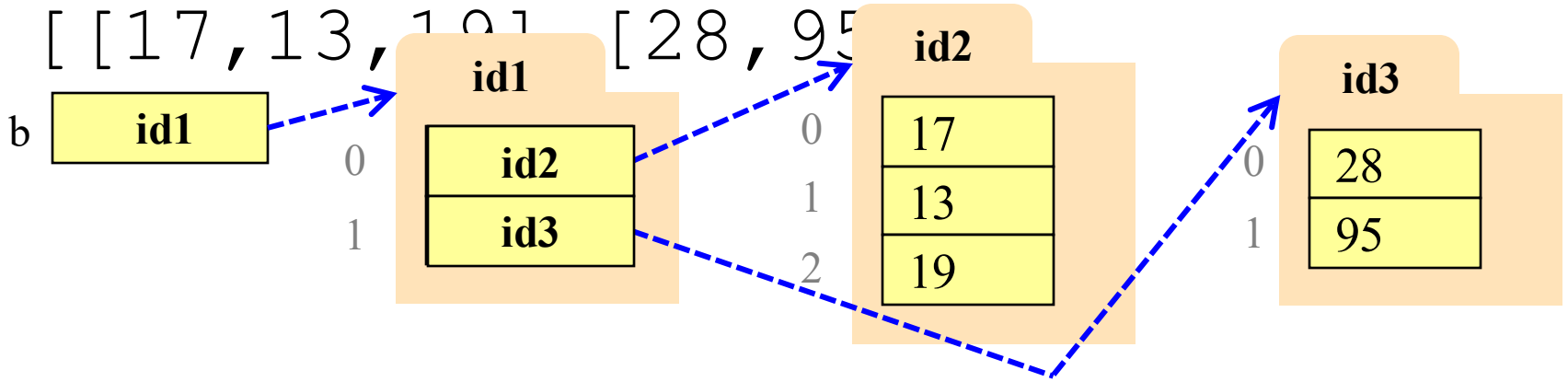
- `b = [[9, 6, 4], [5, 7, 7]]`



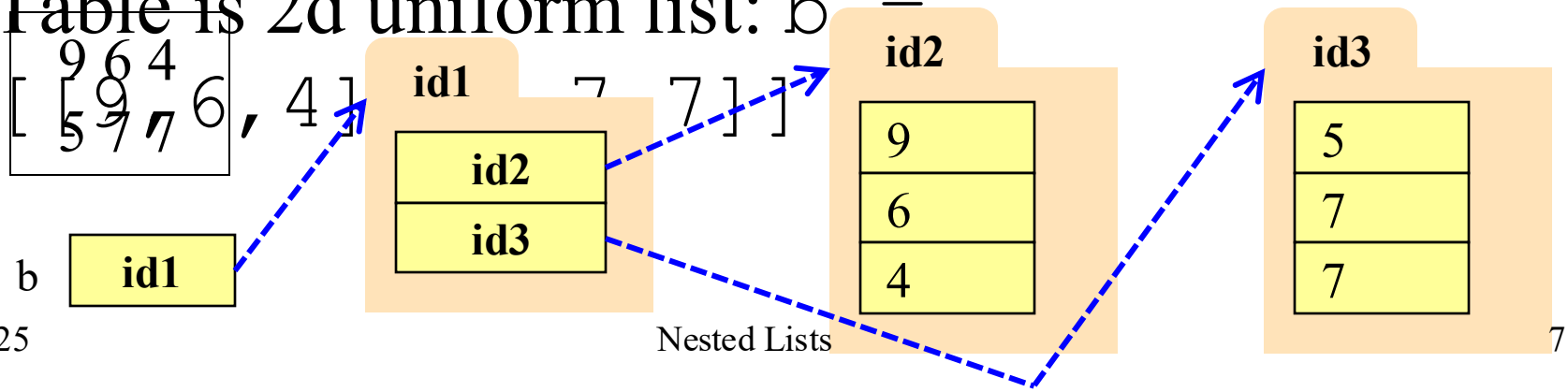
- b holds name of a two-dimensional list
  - Has len(b) elements
  - Its elements are (the names of) 1D lists
- b[i] holds the name of a one-dimensional list (of ints)
  - Has len(b[i]) elements

Ragged Lists vs Tables

- Ragged is 2d uneven list: `b =`



- Table is 2d uniform list: `b =`



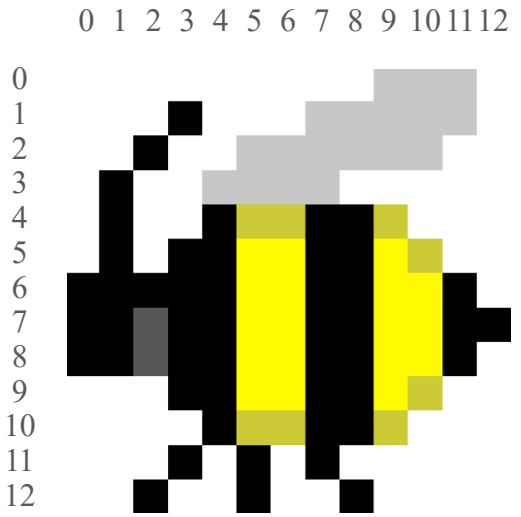
Nested Lists can Represent Tables

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`

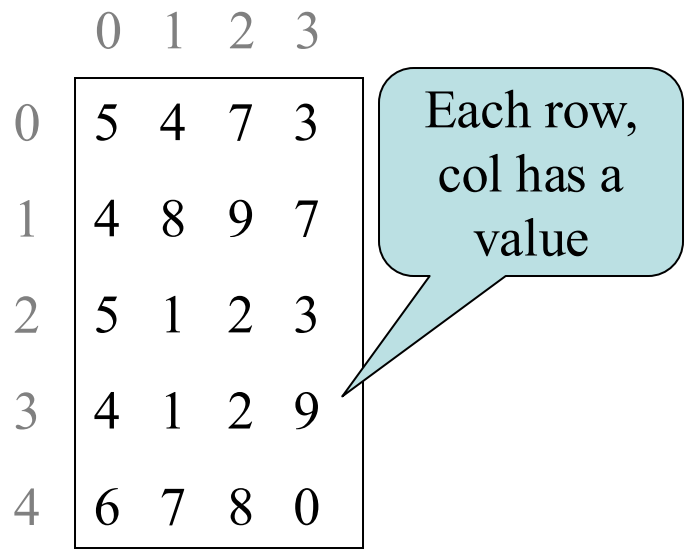
Image



`smile.xlsx`

Representing Tables as Lists

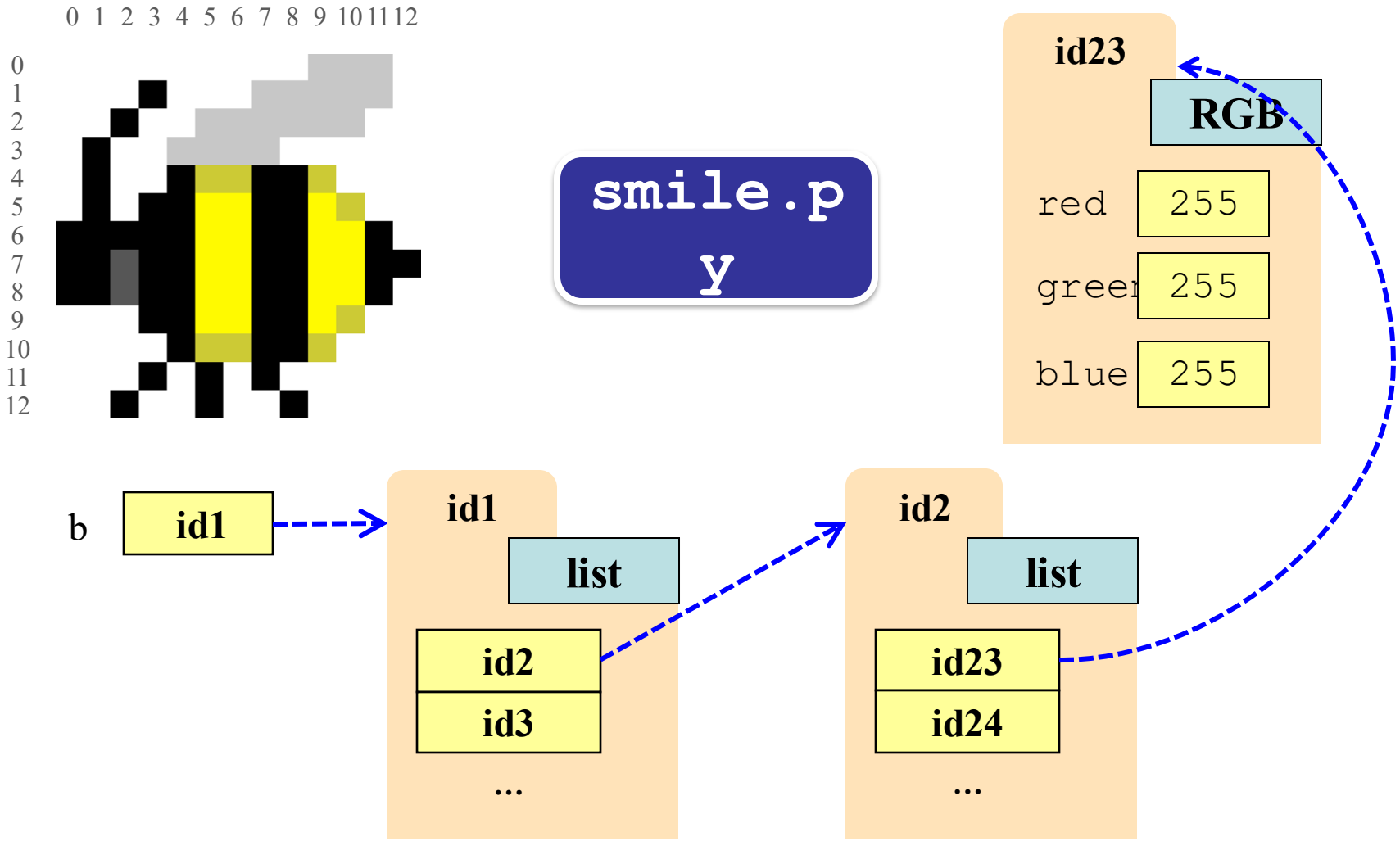
Spreadsheet



- 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`

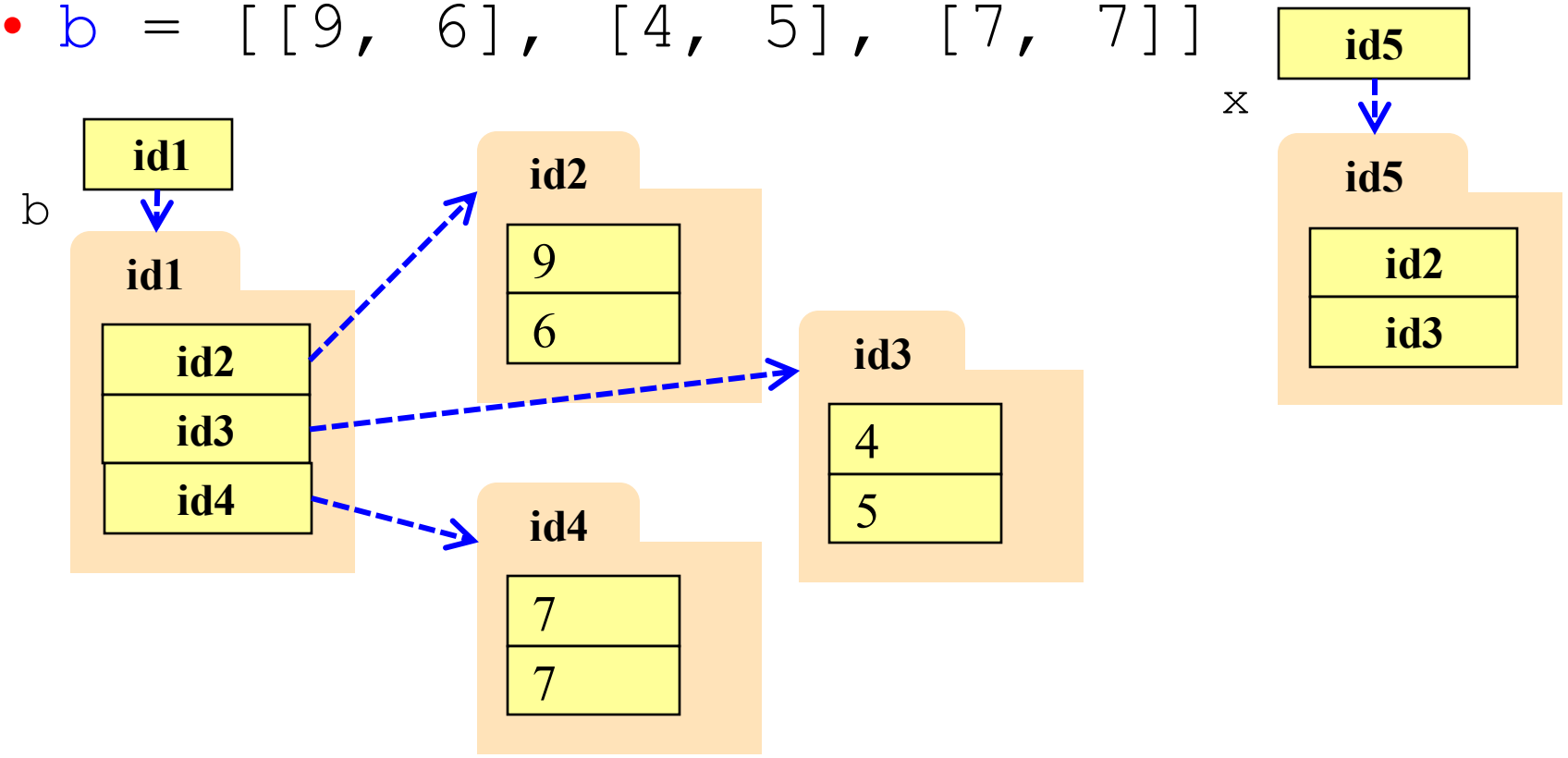
- **An odd symmetry**
  - Number of rows of d: `len(d)`
  - Number of cols in row r of d: `len(d[r])`

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]]`

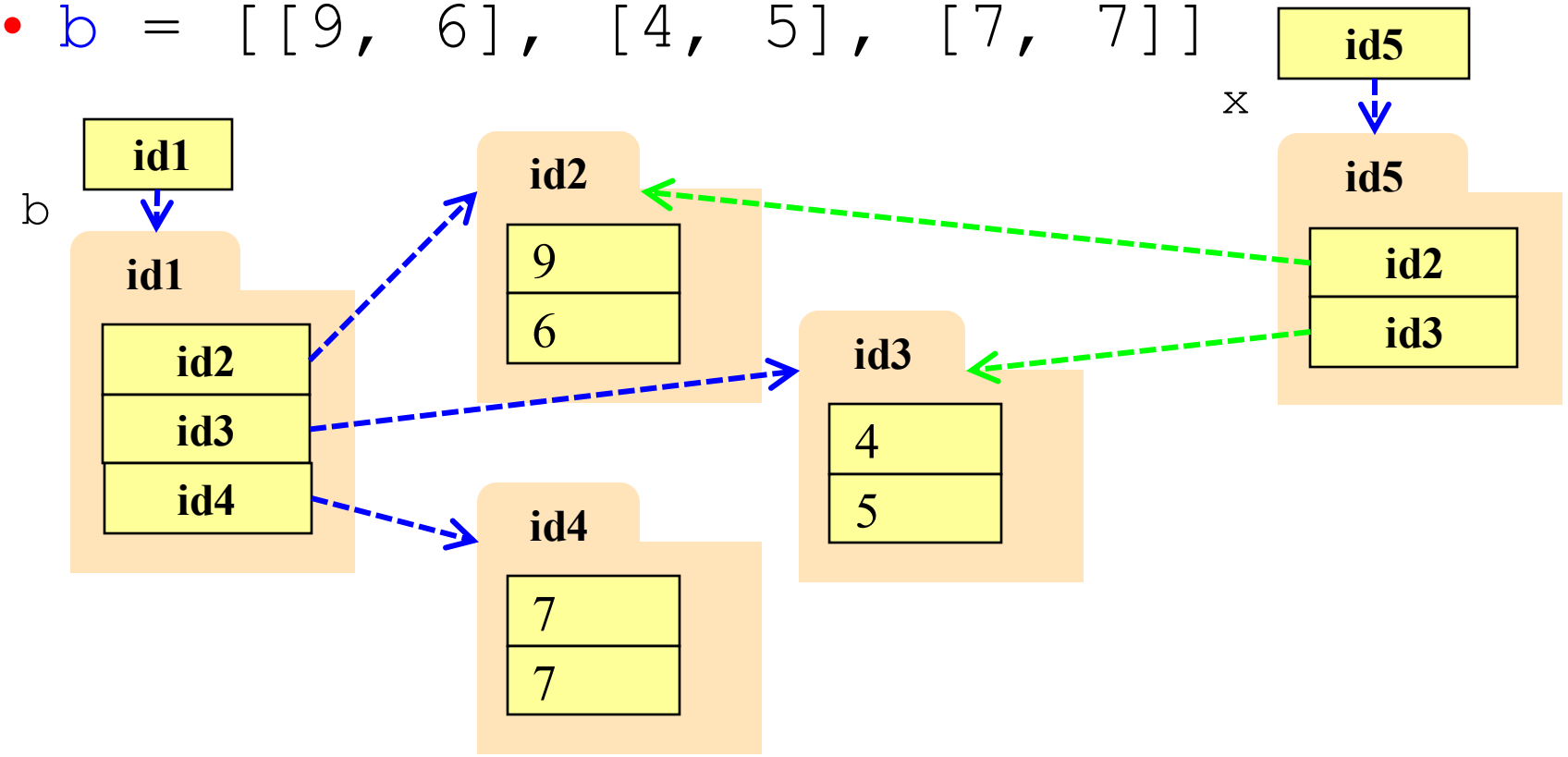
`x = b[:2]`



Slices and Multidimensional Lists

- 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,
```

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

A:
```
[[9,6],[4,5],[7
,7]]
```
B:
```
[[9,6],[4,5,10]
]
```
C:
```
[[9,6],[4,5,10]
,[7,7]]
```

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,
```

- 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]]
```

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
```

Functions over Nested 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 ta
    for row in table
        # Walk through the row
        for item in row:
            if not type(item) in [int,float]:
                result = False
    return result
```

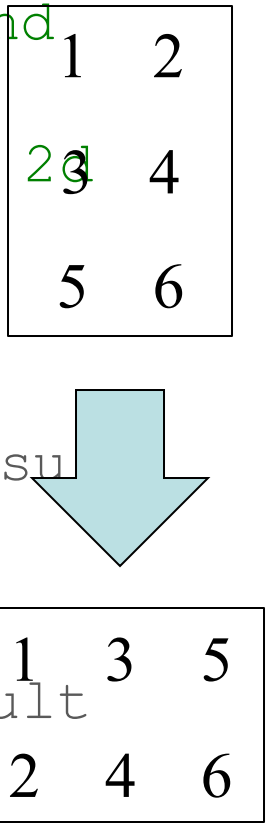
Accumulator

First Loop

Second Loop

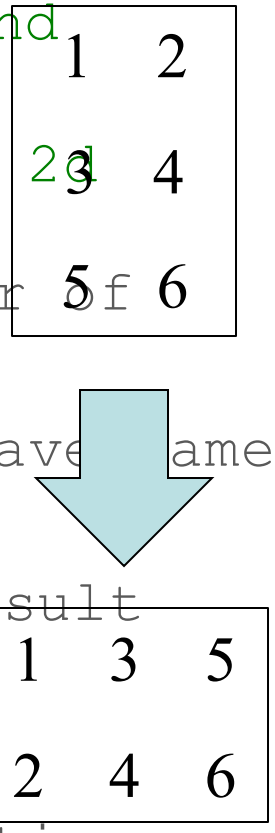
Transpose: A Trickier Example

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

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



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



    return result
```



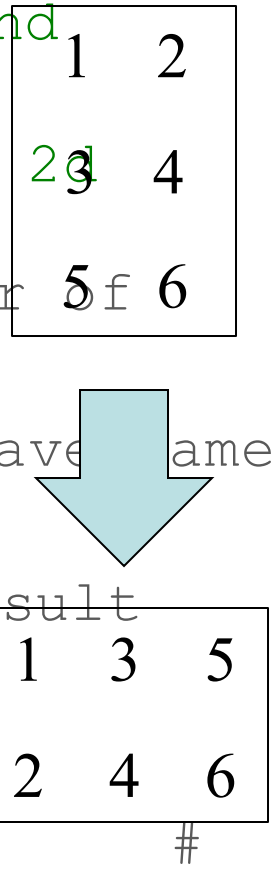
Transpose: A Trickier Example

```
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
```



Transpose: A Trickier Example

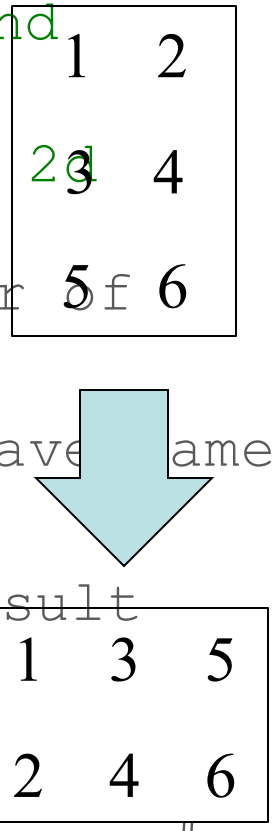
```
def transpose(table):
    """Returns: copy of table with rows and
columns swapped

    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):
```



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
```

Accumulator for each loop

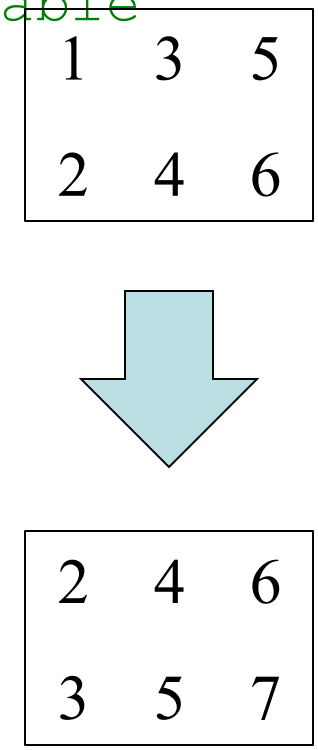


Mutable Functions on Nested Lists

- Function structure similar to that of lists
  - Do not loop over the list (modifying it)
  - Loop over *range of positions* instead
  - No *accumulator* or return statement
- But some important differences
  - May need **multiple for-loops**
  - Depends on if modifying rows or entries
    - **Modifying entries**: two loops
    - **Modifying rows**: could be one or two

A Mutable 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

        # Walk through each column

            # Add 1 to each element

    # No return statement
```

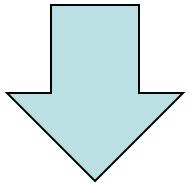


A Mutable Example

```
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
    for rpos in range(
        # Walk through each co  mn
        for cpos in range(len(table[rpos])):
            table[rpos][cpos] =
table[rpos][cpos]+1

    # No return statement
```

Do not loop over the table

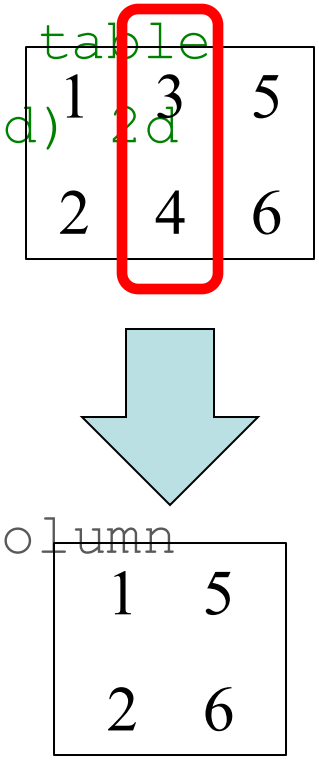
1	3	5
2	4	6



2	4	6
3	5	7

Another Example

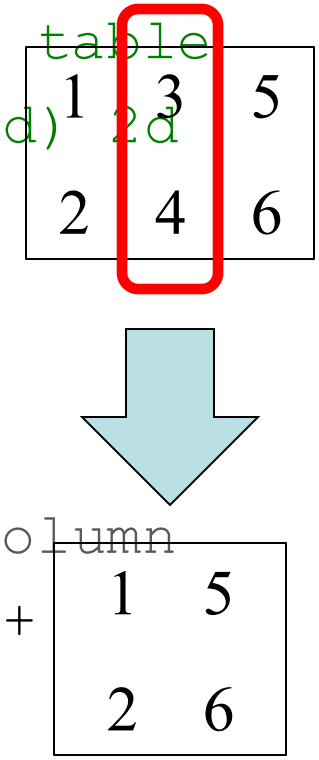
```
def strip(table,col):
    """Removes column col from the given table
    Preconditions: table is a (non-ragged) 2d
List,
    col valid column"""
    # Walk through table
        # Modify each row to slice out column



    # No return statement
```



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 tab
    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



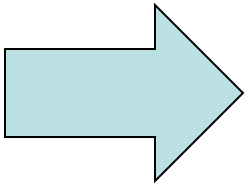
Advanced Topic: Reading CSV Files

- **Recall:** CSV (comma-separate-value) files
  - Portable way of representing tables
  - Supported by Excel and other programs
  - Can also be read as a simple text file
- Possible to load them into Python
  - **Hard way:** Using the Python `csv` module
  - **Easy way:** Using the `introcs` module
- Will revisit hard way later

Using the Function `introcs.read_csv`

- Returns a table of strings (not numbers)
  - Up to *you* to convert the data
  - Similar problem to the `input` function

	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



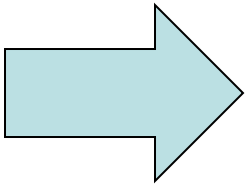
```
[['5',
'4', '7',
'3'],
 ['4',
'8', '9',
'7'],
 ['5',
'1', '2',
'3'],
```

Using the Function `introcs.read_csv`

- Returns a
  - Up to
  - Similar

Can use for-loops to convert values!

	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



```
[['5',
'4', '7',
'3'],
 ['4',
'8', '9',
'7'],
 ['5',
'1', '2',
'3'],
```

Reading Data from a File

```
def load_table(file):
    """Returns: the table stored in the CSV file

    Values are converted to float if possible (remain
as string if not).
    Precondition: file is the name of a CSV file"""
    # Load the file into a table

    # Access each row of the table

        # Access each element in each row

            # Convert to float if possible



    # Return the result
```

Reading Data from a File

```
def load_table(file):
    """Returns: the table stored in the CSV file

    Values are converted to float if possible (remain
as string if not).

    Precondition: file is the name of a CSV file"""
    # Load the file into a table
    result = introcs.read_csv(fil
    # Access each row of the t

        # Access each element

            # Convert to float if possible

    return result
```

This part is mutable

Reading Data from a File

```
def load_table(file):
    """Returns: the table stored in the CSV file

    Values are converted to float if possible (remain
as string if not).

    Precondition: file is the name of a CSV file"""
    # Load the file into a table
    result = introcs.read_csv(file)
    # Access each row of the table
    for r in range(len(result)):
        # Access each element in each row
        for c in range(len(result[r])):
            # Convert to float if possible
            try:
                result[r][c] = float(result[r][c])
            except:
                pass
    return result
```

Writing to a CSV File

- You can also write a table to a CSV file
  - **Function:** `introcs.write_csv(table,filename)`
  - File is written to your current directory
  - But there are restrictions on the table!
- Table must be a rectangular, 2d-list
- The first row must be the header
  - Header = the titles of each column
  - So all values must be strings!
- Later rows can be any type
  - Function will convert them using `str`