Lecture 16

Nested Lists and Dictionaries

Announcements for This Lecture

Prelim and Regrades

- Regrades are now open
  - Only for MAJOR mistakes
  - You might *lose* points
- The regrade process
  - Ask in Gradescope
  - Tell us what to look for
  - If valid, we will respond
  - We will also update CMS

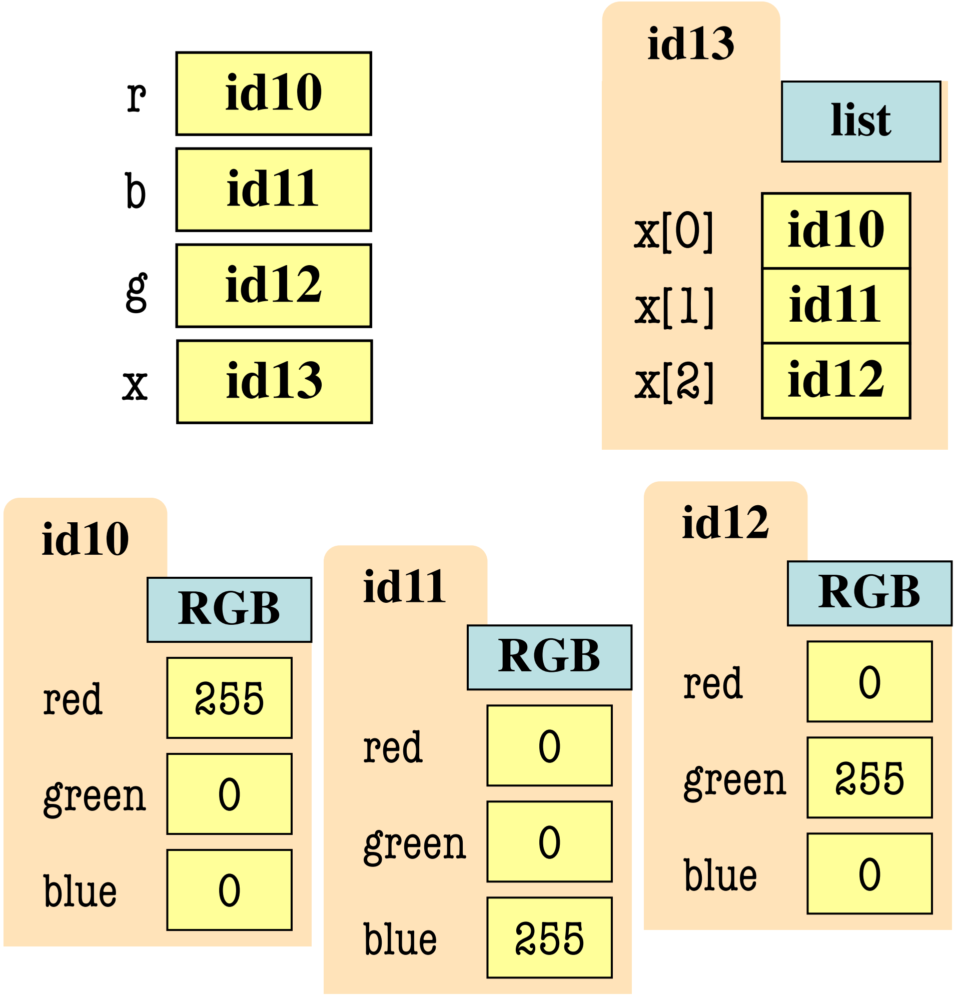
Assignments/Reading

- Should be working on A4
  - Tasks 1-2 by tomorrow
  - Task 3 by the weekend
  - Recursion next week
- **Reading**: Chapters 15, 16
  - Chapter 17 for next week
  - Lot of potential reading
  - … but we are covering a lot

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]
```



r id10
b id11
g id12
x id13

id13
list
x[0] id10
x[1] id11
x[2] id12

id10
RGB
red 255
green 0
blue 0

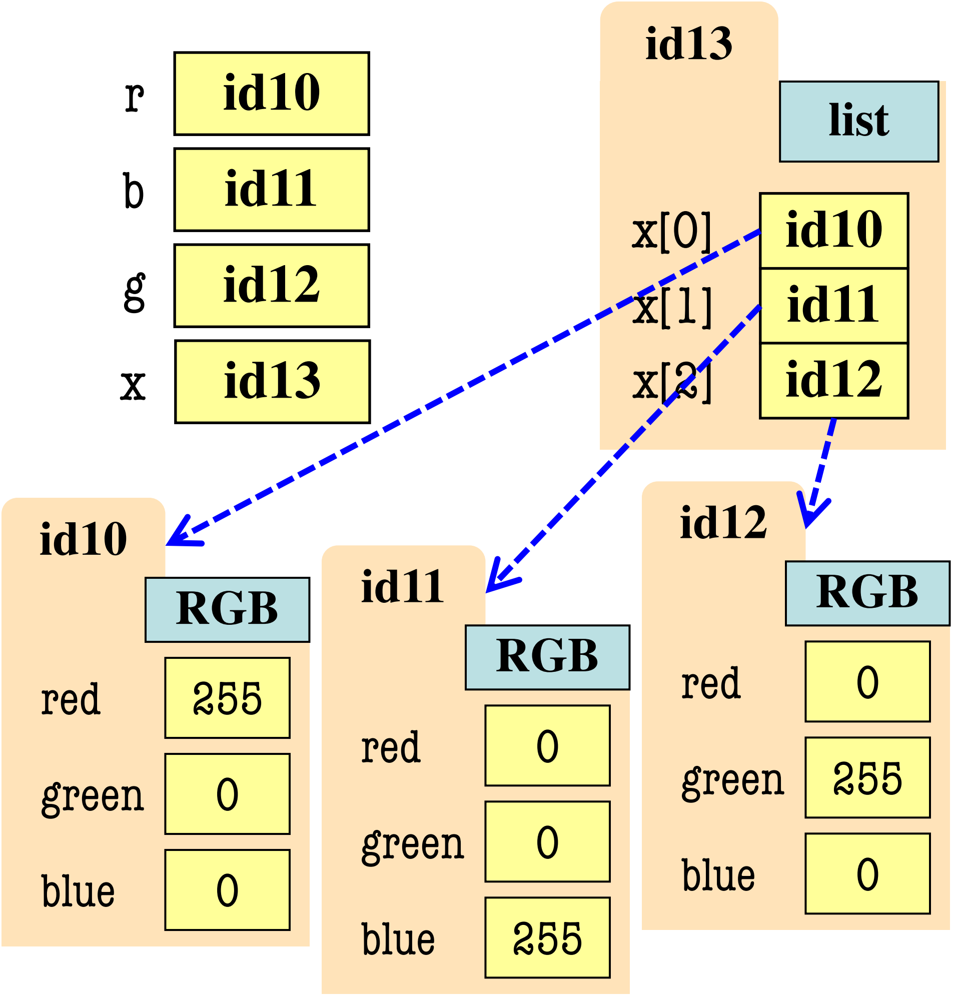
id11
RGB
red 0
green 0
blue 255

id12
RGB
red 0
green 255
blue 0

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:

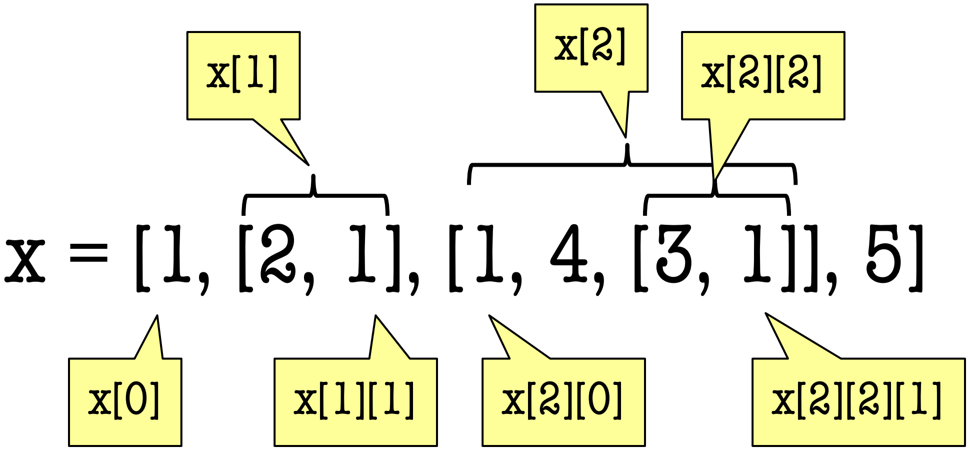
```
>>> r = introcs.RGB(255,0,0)
>>> b = introcs.RGB(0,0,255)
>>> g = introcs.RGB(0,255,0)
>>> x = [r,b,g]
```



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]
```



x[1]

x[2]

x[2][2]

```
x = [1, [2, 1], [1, 4, [3, 1]], 5]
```

x[0]

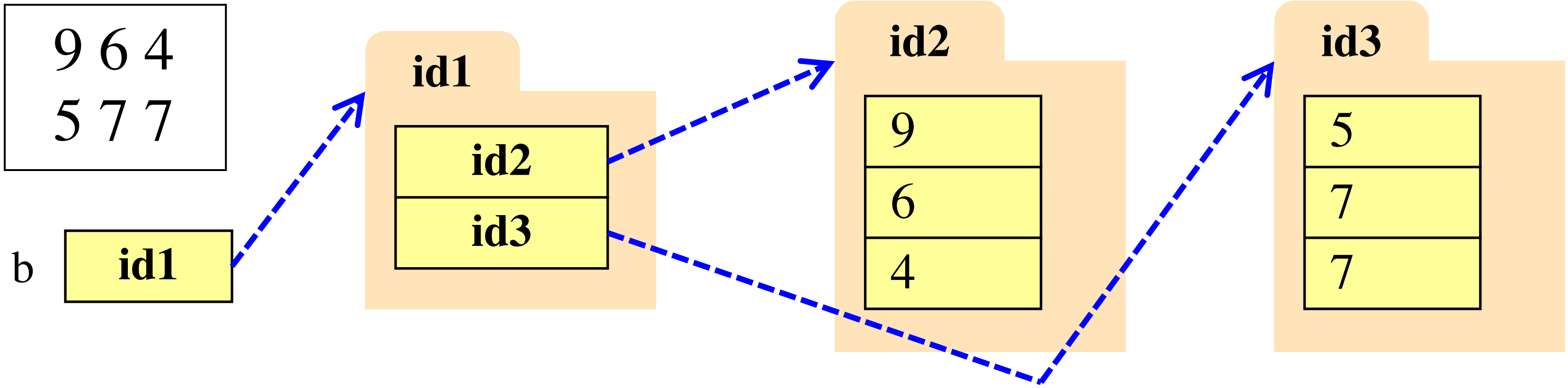
x[1][1]

x[2][0]

x[2][2][1]

How Multidimensional Lists are Stored

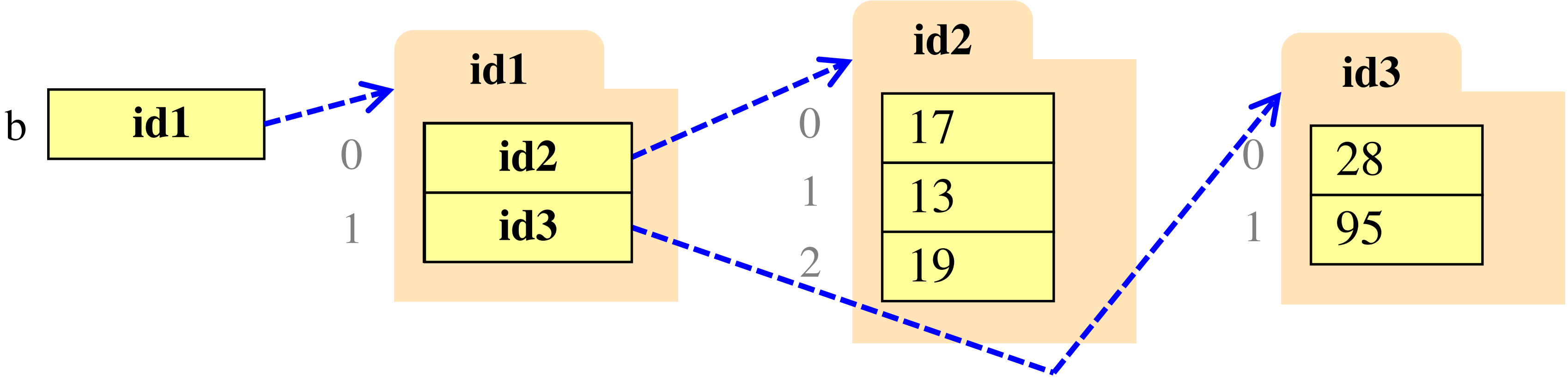
- `b = [[9, 6, 4], [5, 7, 7]]`



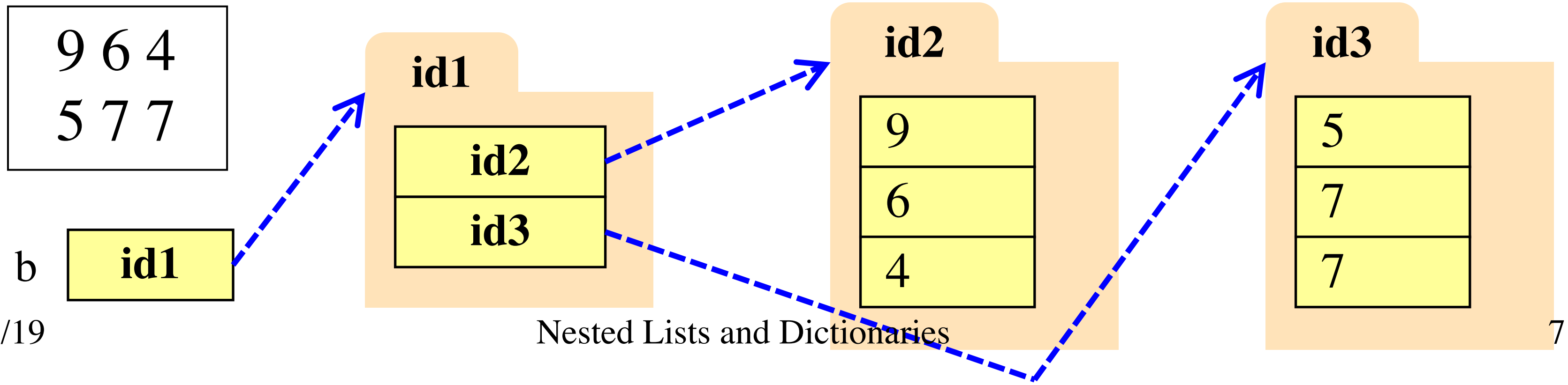
- b holds name of a two-dimensional list
  - Has len(b) elements
  - Its elements are (the names of) 1D lists
- b[i] holds the name of a one-dimensional list (of ints)
  - Has len(b[i]) elements

Ragged Lists vs Tables

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



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


9 6 4
5 7 7

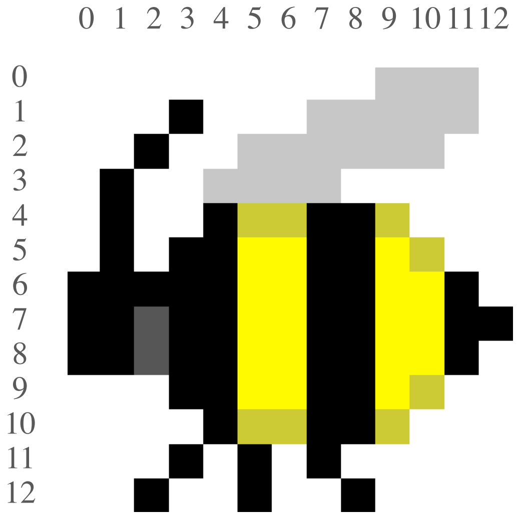
Nested Lists can Represent Tables

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

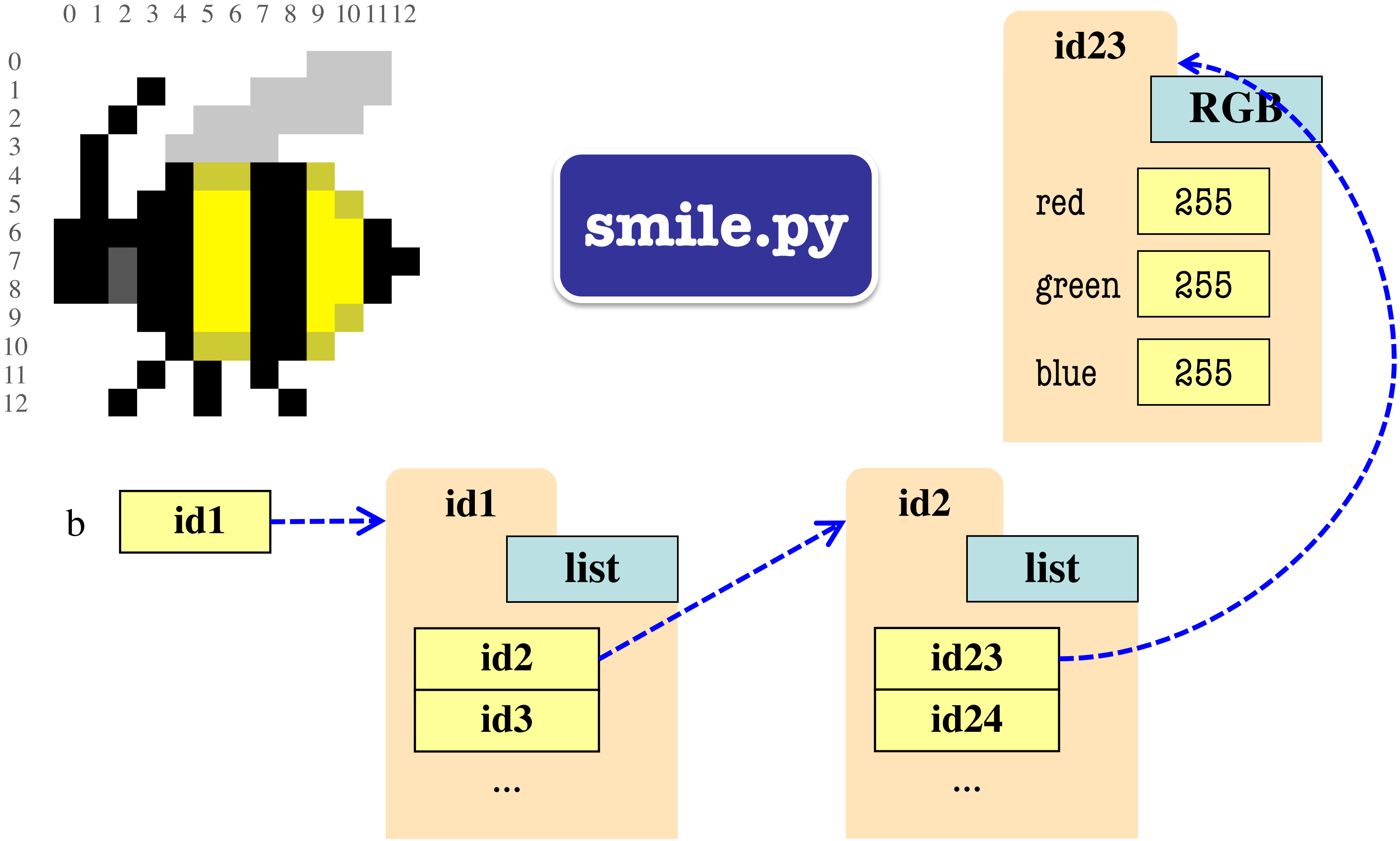
	0	1	2	3
0	5	4	7	3
1	4	8	9	7
2	5	1	2	3
3	4	1	2	9
4	6	7	8	0

Each row, col has a value

- 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



smile.py

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`

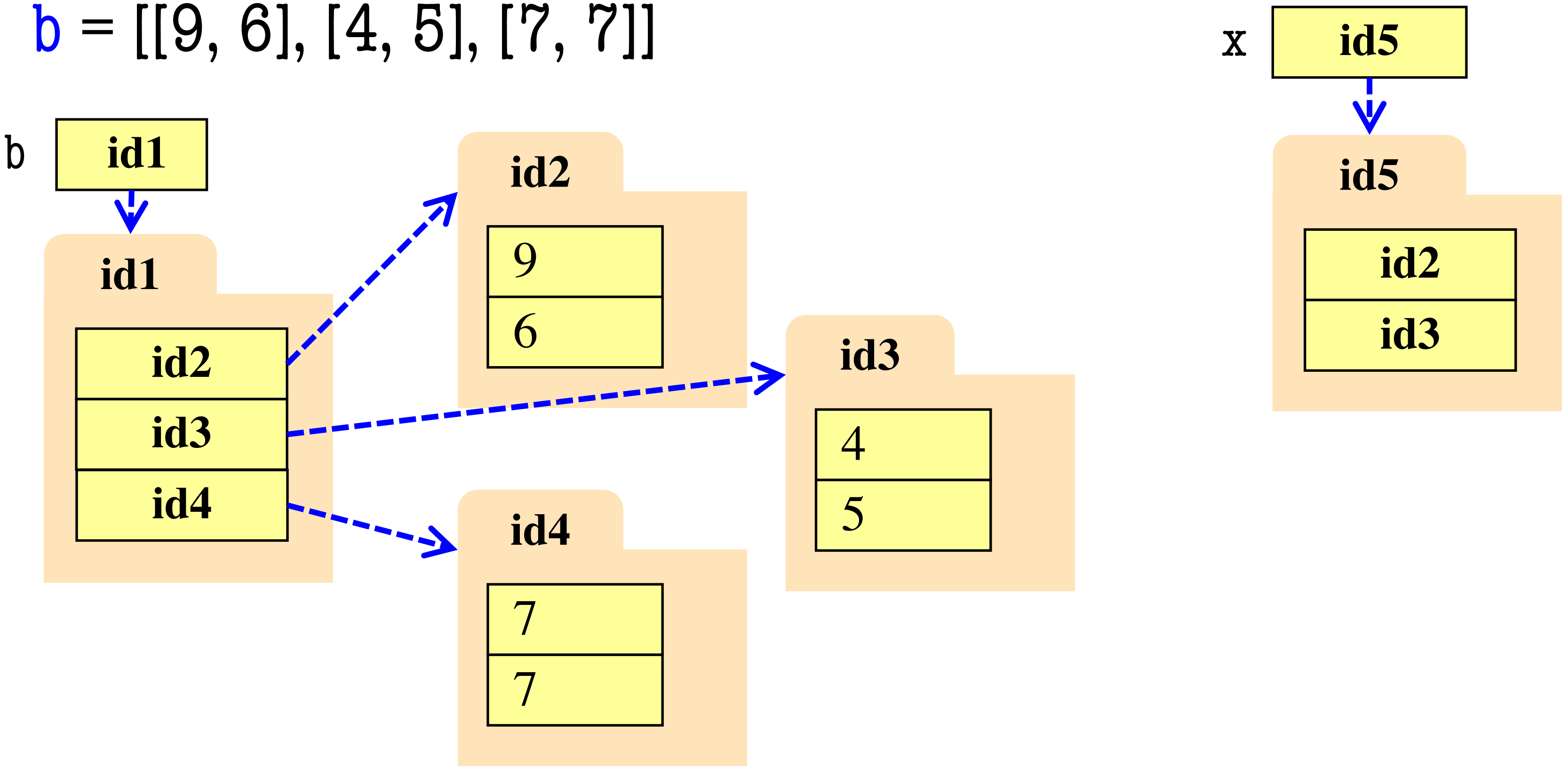
- **An odd symmetry**
  - Number of rows of d: `len(d)`
  - Number of cols in row r of d: `len(d[r])`

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]]`

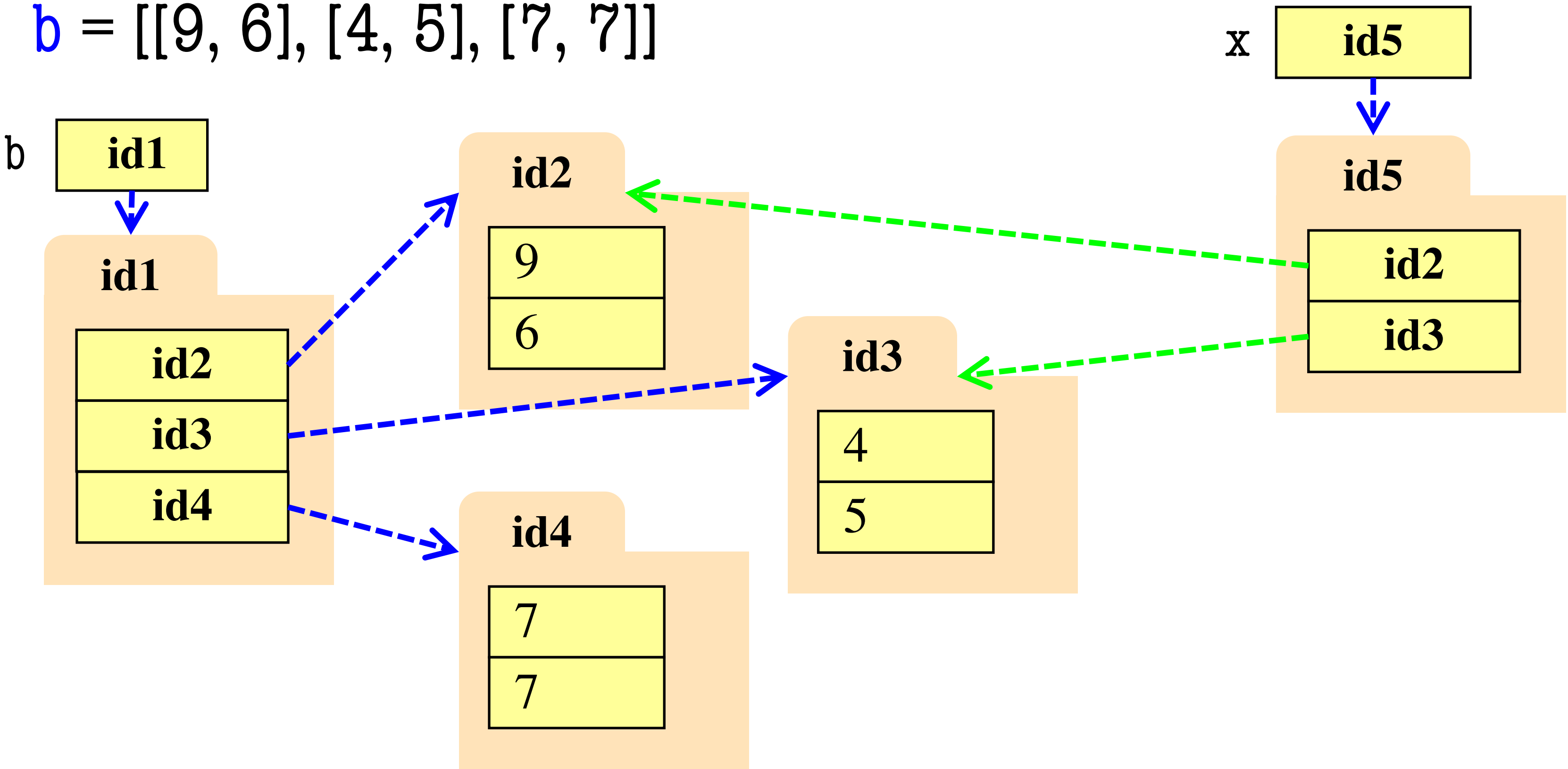
`x = b[:2]`



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, 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

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
```

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

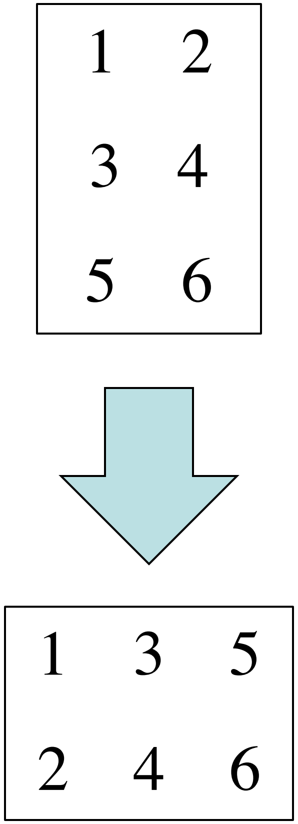
Accumulator

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



    return result
```

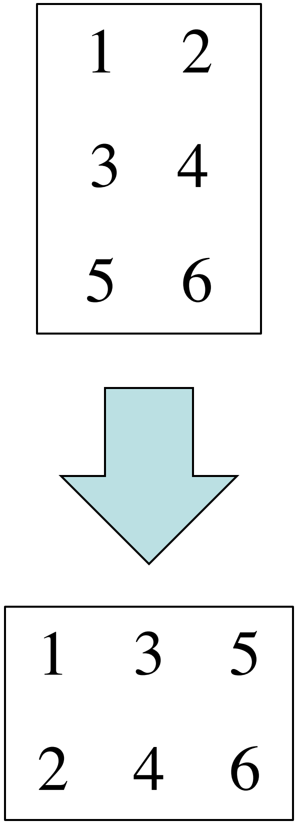


1	2
3	4
5	6

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):
        # Get the column elements at position m

        # Make a new list for this column

        # Add this row to accumulator table


    return result
```

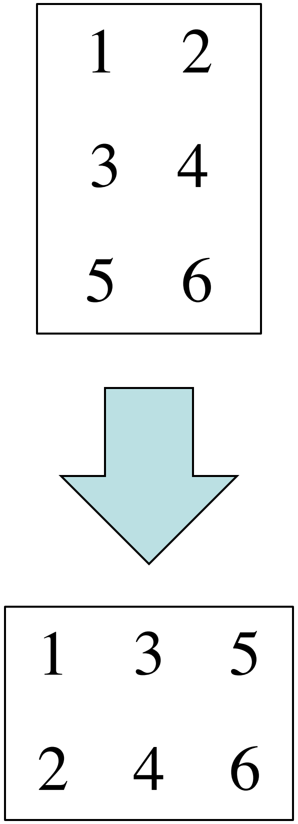


1	2
3	4
5	6

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
```



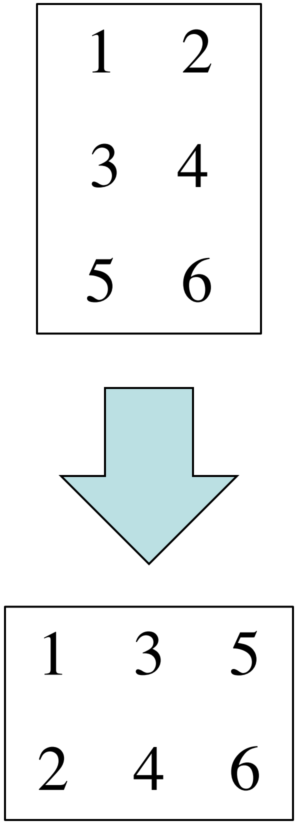
1	2
3	4
5	6

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 = []                          ) 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

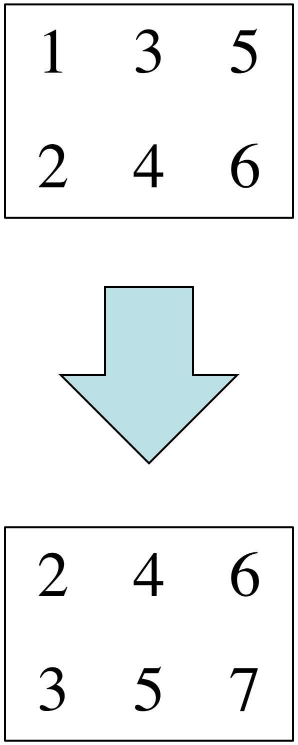


1	2
3	4
5	6

1	3	5
2	4	6

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
```



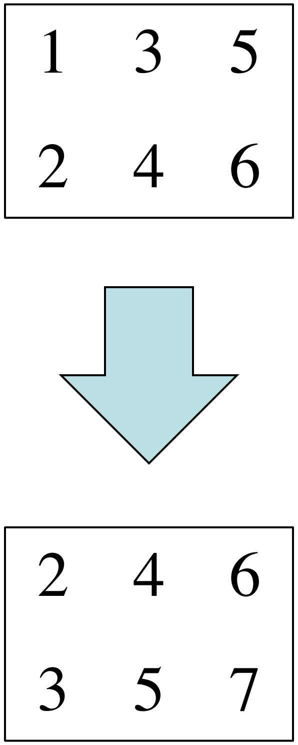
1	3	5
2	4	6

2	4	6
3	5	7

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
    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

Key-Value Pairs

- The last built-in type: dictionary (or `dict`)
  - One of the most important in all of Python
  - Like a list, but built of key-value pairs
- **Keys:** Unique identifiers
  - Think social security number
  - At Cornell we have netids: jrs1
- **Values:** Non-unique Python values
  - John Smith (class ’13) is jrs1
  - John Smith (class ’16) is jrs2

Idea: Lookup values by keys

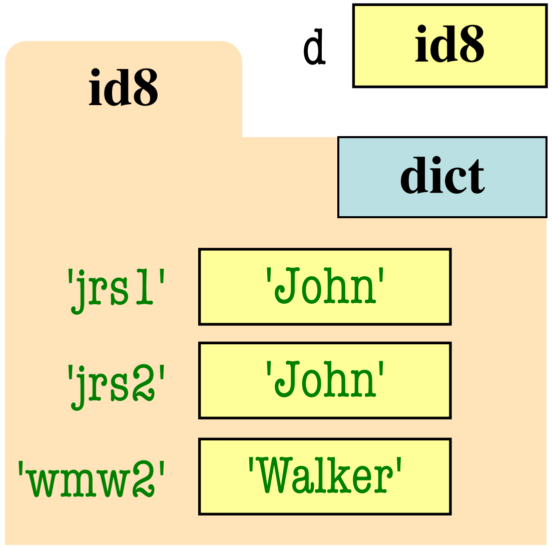
Basic Syntax

- Create with format: {k1:v1, k2:v2, ...}
  - Both keys and values must exist
  - **Ex:** d={'jrs1':'John','jrs2':'John','wmw2':'Walker'}
- **Keys** must be **non-mutable**
  - ints, floats, bools, strings, tuples
  - **Not** lists or custom objects
  - Changing a key's contents hurts lookup
- **Values** can be **anything**

Using Dictionaries (Type dict)

- Access elts. like a list
  - d['jrs1'] evals to 'John'
  - d['jrs2'] does too
  - d['wmw2'] evals to 'Walker'
  - d['abc1'] is an **error**
- Can test if a key exists
  - 'jrs1' in d evals to True
  - 'abc1' in d evals to False
- But cannot slice ranges!

```
d = {'js1':'John','js2':'John',
    'wmw2':'Walker'}
```

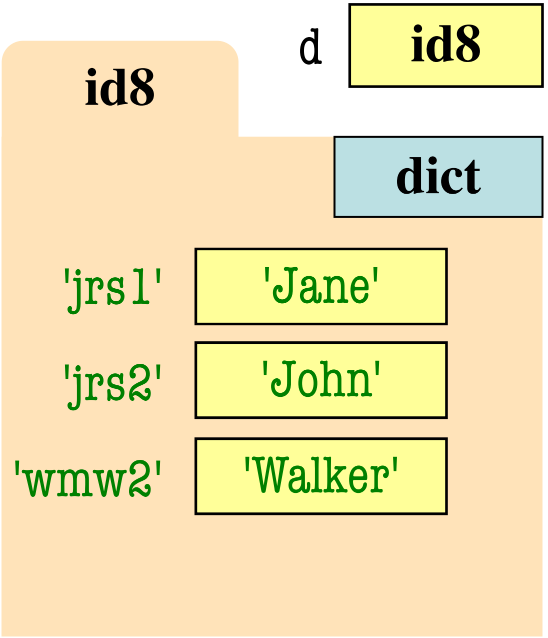


Key-Value order in folder is not important

Dictionaries Can be Modified

- **Can reassign values**
  - d['jrs1'] = 'Jane’
  - Very similar to lists
- Can add new keys
  - d[‘aaa1'] = 'Allen’
  - Do not think of order
- Can delete keys
  - del d['wmw2’]
  - Deletes both key, value

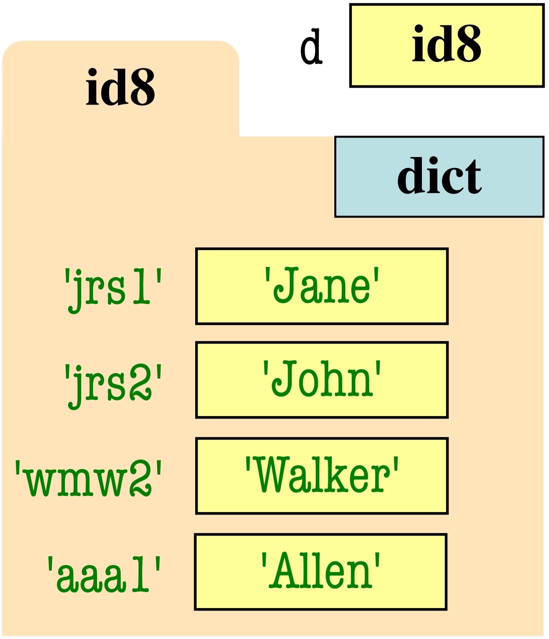
```
d = {'jrs1':'John','jrs2':'John',
     'wmw2':'Walker'}
```



Dictionaries Can be Modified

- Can reassign values
  - d['jrs1'] = 'Jane’
  - Very similar to lists
- **Can add new keys**
  - d[‘aaa1'] = 'Allen’
  - Do not think of order
- Can delete keys
  - del d['wmw2’]
  - Deletes both key, value

```
d = {'jrs1':'John','jrs2':'John',
     'wmw2':'Walker'}
```

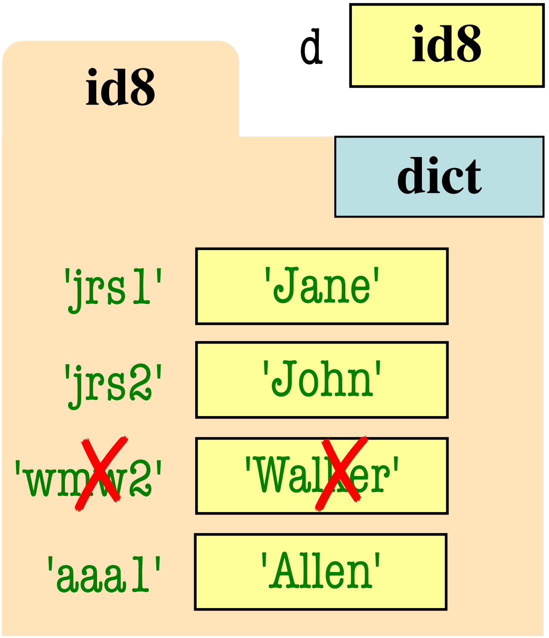


Dictionaries Can be Modified

- Can reassign values

Change key = Delete + Add

  - d['jrs1'] jrs2':'John', wmw2':'Walker'}
  - Very similar to lists
- Can add new keys
  - d['aaa1'] = 'Allen'
  - Do not think of order
- **Can delete keys**
  - del d['wmw2']
  - Deletes both key, value



Nesting Dictionaries

- Remember, values can be anything
  - Only restrictions are on the keys
- Values can be lists (**Visualizer**)
  - `d = {'a':[1,2], 'b':[3,4]}`
- Values can be other dicts (**Visualizer**)
  - `d = {'a':{'c':1,'d':2}, 'b':{'e':3,'f':4}}`
- Access rules similar to nested lists
  - **Example:** `d['a']['d'] = 10`

Example: JSON File

```
{
  "wind" : {
    "speed" : 13.0,
    "crosswind" : 5.0
    },
  "sky" : [
    {
      "cover" : "clouds",
      "type" : "broken",
      "height" : 1200.0
    },
    {
      "type" : "overcast",
      "height" : 1800.0
    }
  ]
}
```

Nested Dictionary

Nested List

Nested Dictionary

- **JSON:** File w/ Python dict
  - Actually, minor differences
- `weather.json`:
  - Weather measurements at Ithaca Airport (2017)
  - **Keys**: Times (Each hour)
  - **Values**: Weather readings
- This is a *nested* JSON
  - Values are also dictionaries
  - Containing more dictionaries
  - And also containing lists

Dictionaries: Iterable, but not Sliceable

- Can loop over a dict
  - Only gives you the keys
  - Use key to access value

```
for k in d:
    # Loops over keys
    print(k)       # key
    print(d[k])  # value
```

- Can iterate over values
  - **Method:** d.values()
  - But no way to get key
  - Values are not unique

```
# To loop over values only
for v in d.values():
    print(v)       # value
```

Other Iterator Methods

- **Keys:** d.keys()
  - Sames a normal loop
  - Good for *extraction*
  - keys = list(d.keys())

```
for k in d.keys():
    # Loops over keys
    print(k)      # key
    print(d[k])   # value
```

- **Items:** d.items()
  - Gives key-value pairs
  - Elements are tuples
  - Specialized uses

```
for pair in d.items():
    print(pair[0]) # key
    print(pair[1]) # value
```

Other Iterator Methods

- **Keys:** d.keys()
  - Sames a normal loop
  - Good for *extraction*
  - keys

```
for k in d.keys():
    # Loops over keys
    print(k)      # key
    print(d[k])   # value
```

So mostly like loops over lists

- **Items:** d.items()
  - Gives key-value pairs
  - Elements are tuples
  - Specialized uses

```
for pair in d.items():
    print(pair[0]) # key
    print(pair[1]) # value
```

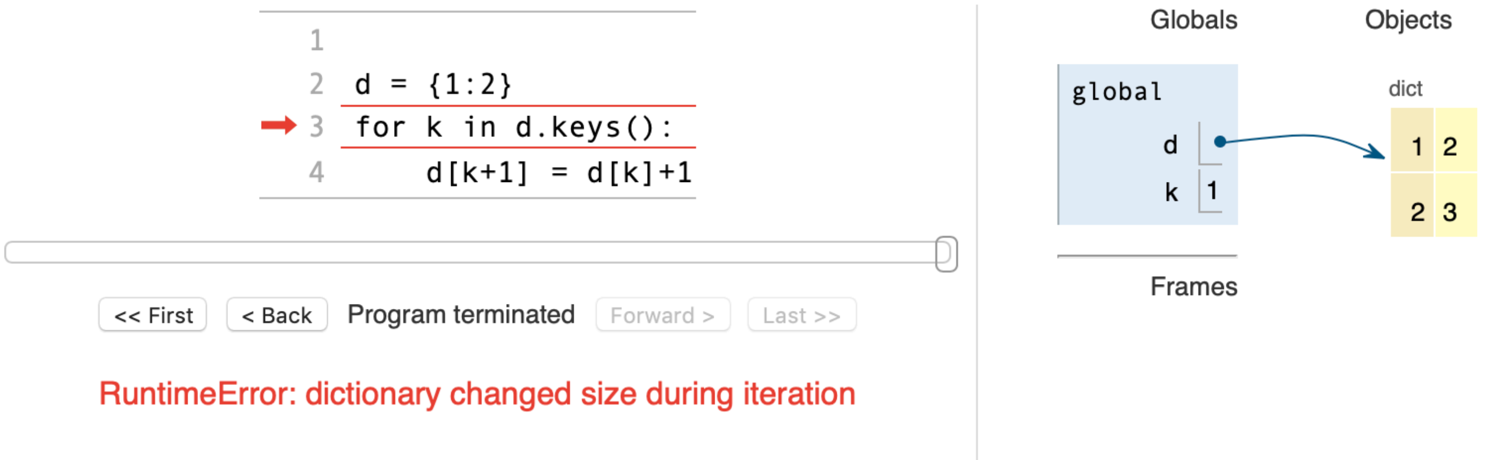
Dictionary Loop with Accumulator

```
def max_grade(grades):
    """Returns max grade in the grade dictionary

    Precondition: grades has netids as keys, ints as values"""
    maximum = 0                    # Accumulator
    # Loop over keys
    for k in grades:
        if grades[k] > maximum:
            maximum = grades[k]

    return maximum
```

Mutable Dictionary Loops

- Restrictions are different than list
  - Okay to loop over dictionary being changed
  - You are looping over *keys*, not *values*
  - Like looping over positions
- But you **may not add or remove** keys!
  - Any attempt to do this will fail
  - Have to create a key list if you want to do

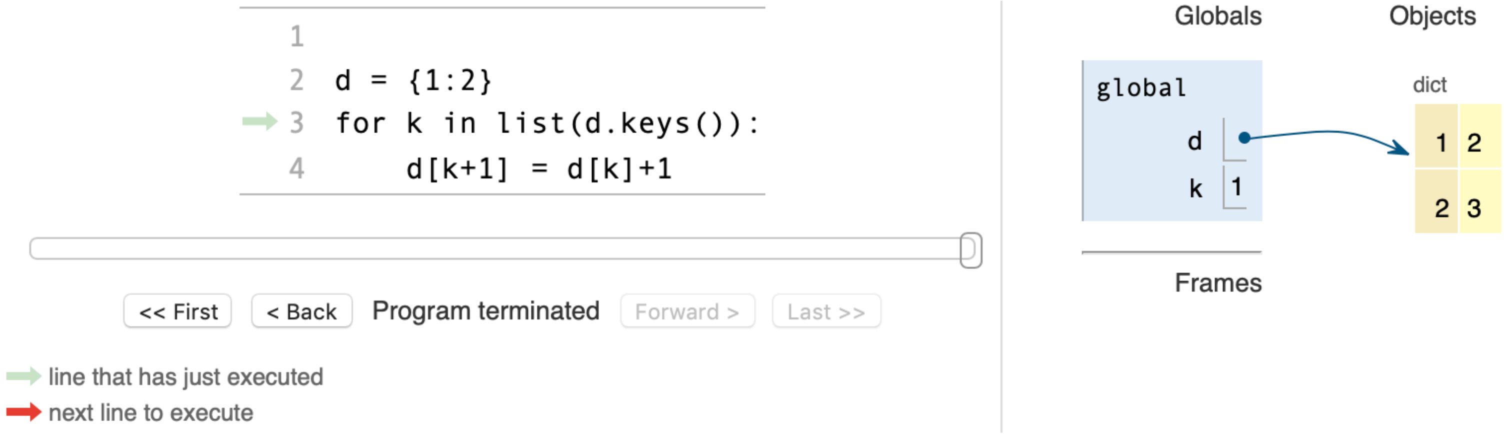
A Subtle Difference



```
d = {1:2}
for k in d.keys():
    d[k+1] = d[k]+1
```

Program terminated

RuntimeError: dictionary changed size during iteration



```
d = {1:2}
for k in list(d.keys()):
    d[k+1] = d[k]+1
```

Program terminated

But This is Okay

```
def give_extra_credit(grades,netids,bonus):
    """Gives bonus points to everyone in sequence netids

    Precondition: grades has netids as keys, ints as values.
    netids is a sequence of strings that are keys in grades
    bonus is an int."""
    # No accumulator.  This is a procedure

    for student in grades:
        if student in netids:              # Test if student gets a bonus
            grades[student] = grades[student]+bonus
```

Could also loop over **netids**

Appendix: Tuple Expansion

Optional Topic not in Lecture

- This topic **is never used in class**
  - Not in any lab or assignment
  - Not on any exam (prelim 2 or final)
- This topic **is never mentioned in lecture**
  - These slides are your only introduction
  - As well as some source-code demos
- This topic is **only for interested students**
  - We get a lot of requests about it

Tuple Expansion

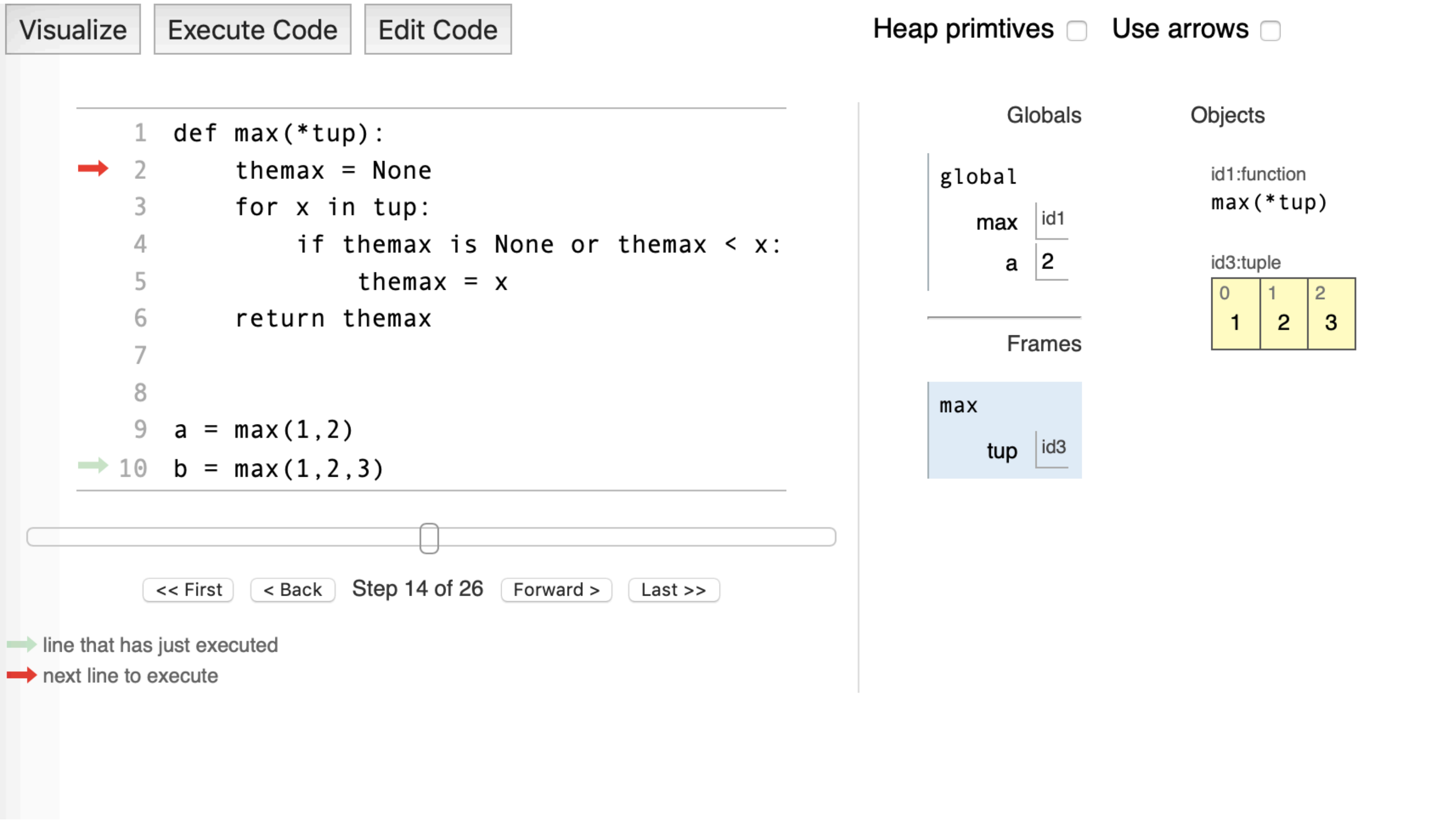
- Last use of lists/tuples is an advanced topic
  - But will see if read Python code online
  - Favored tool for data processing
- Observation about function calls
  - Function calls look like name + tuple
  - Why not pass a *single* argument: the tuple?
- Purpose of tuple expansion: `*tuple`
  - But only works in certain **contexts**

Tuple Expansion Example

```
>>> def add(x, y)
...       """Returns x+y """
...       return x+y
...
>>> a = (1,2)
>>> add(*a)               # Slots each element of a into params
3
>>> a = (1,2,3)           # Sizes must match up
>>> add(*a)
ERROR
```

Have to use in **function call**

Also Works in Function Definition



```
def max(*tup):
    themax = None
    for x in tup:
        if themax is None or themax < x:
            themax = x
    return themax


a = max(1,2)
b = max(1,2,3)
```

Step 14 of 26

→ line that has just executed

→ next line to execute

Globals

global

max id1

a 2

Frames

max

tup id3

Objects

id1:function

max(*tup)

id3:tuple

0	1	2
1	2	3

Also Works in Function Definition

```
def max(*tup):
    """Returns the maxi

    Param tup: The tuple of numbers
    Precond: Each element of tup is an int or float"""
    themax = None
    for x in tup:
        if themax == None or themax < x:
            themax = x
    return themax
```

Automatically converts all arguments to tuple

Why Bring this Up Now?

- We were talking about lists
  - This is technically tuple, not list, expansion
- But can be done with any *sequence*
  - The sliceable types: tuple, string, list
  - **Example:** function(*'string')
- Common to see expansion **calls** done with lists
  - People prefer lists over tuples (for mutability)
- But always a tuple in function **definition**
  - Even if pass *'string' as argument