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


1

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

2

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

- 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]]
```

3

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

4

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]

5

Functions on Nested Lists

```
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
```


6

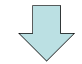
Transpose: A Trickier Example

```
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(num
        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

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

8

Basic Syntax

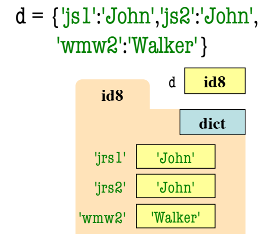
- Create with format: {k1:v1, k2:v2, …}
  - Both keys and values must exist
  - **Ex:** d={'jrs1':'John','jrs2':'John','wmw2':'Walker'}
- **Keys** must be **non-mutable**
  - ints, floats, bools, strings, tuples
  - **Not** lists or custom objects
  - Changing a key's contents hurts lookup
- **Values**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

9

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'}



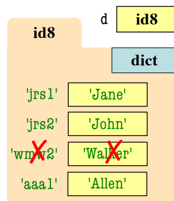
Key-Value order in folder is not important

10

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'}



11

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
```

12