



<http://www.cs.cornell.edu/courses/cs1110/2022sp>

Lecture 12: Nested Lists and Dictionaries

(Sections 11.1-11.5)

CS 1110

Introduction to Computing Using Python

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]



Lecture Afterthoughts

- Someone asked about iterating through the elements of a dict so I've added a new slide about this (slide 12)

Announcements

- TA Office Hours: *Come to them!*
 - *Ask any question... or don't even ask a question!*
 - *"I don't know what's going on..." ← say this!*
 - **Attend early!**
- A3 is out
 - Show your code only to your partner or course staff
 - Look only at the code of your partner
- Masking policy:
 - Keep wearing masks in non-distanced educational settings: this does mean all in-person office/consulting hours

Dictionaries (Type `dict`)

Description

- List of **key-value** pairs
 - Keys are unique
 - Values need not be
- Example: net-ids
 - net-ids are **unique** (a key)
 - names need not be (values)
 - js1 is John Smith (class '13)
 - js2 is John Smith (class '16)

Python Syntax

- Create with format:

```
{key1:value1,  
key2:value2, ...}
```
- Keys must be **immutable**
 - ints, floats, bools, strings
 - **Not** lists or custom objects
- Values can be anything
- Example:

```
d = {'js1': 'John Smith',  
     'js2': 'John Smith',  
     'tm55': 'Toni Morrison'}
```

Using Dictionaries (Type dict)

```
>>> d = {'ec1': 'Ezra', 'ec2': 'Ezra', 'tm55': 'Toni'}
```

```
>>> d['ec1']
```

```
'Ezra'
```

```
>>> d[0]
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>
```

```
KeyError: 0
```

```
>>> d[:1]
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>
```

```
TypeError: unhashable type: 'slice'
```

```
>>>
```

Global Space

d id8

Heap Space

id8

dict

'ec1'

'Ezra'

'ec2'

'Ezra'

'tm55'

'Toni'

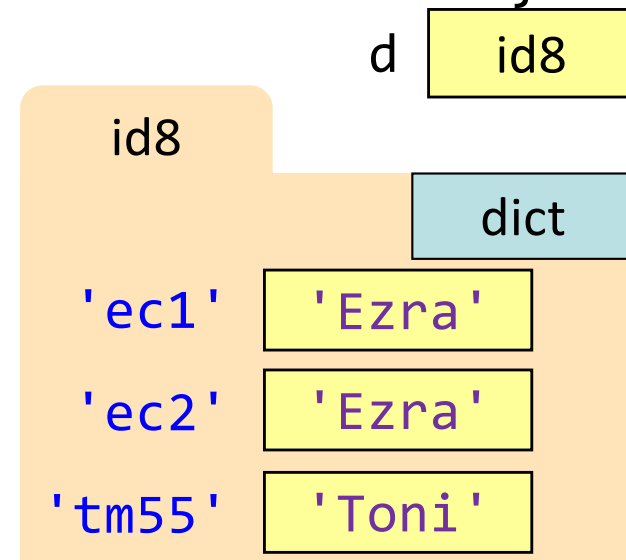
- Can access elements like a list, but...
 - Must use the key, not an index
 - Cannot slice ranges

Dictionaries are **mutable** (1, before)

1. Can reassign values

- `d['ec1'] = 'Ellis'`

```
d = {'ec1': 'Ezra',  
     'ec2': 'Ezra',  
     'tm55': 'Toni'}
```

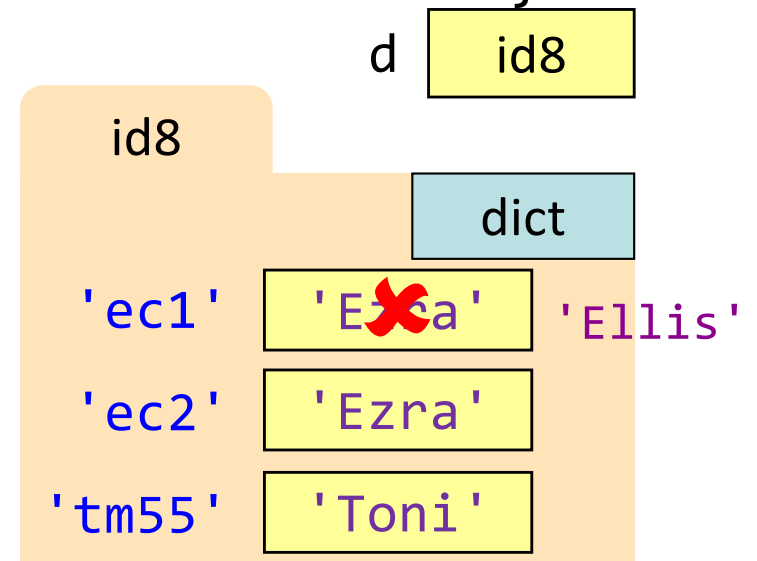


Dictionaries are **mutable** (1, after)

1. Can reassign values

- `d['ec1'] = 'Ellis'`

```
d = {'ec1': 'Ezra',  
     'ec2': 'Ezra',  
     'tm55': 'Toni'}
```



Dictionaries are **mutable** (2, before)

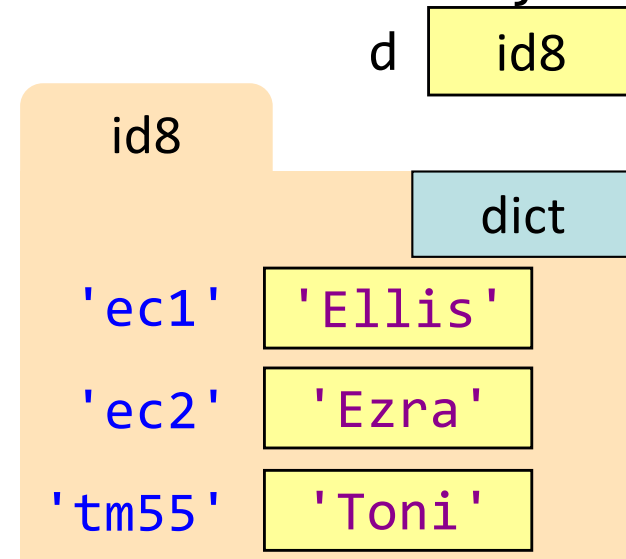
1. Can reassign values

- `d['ec1'] = 'Ellis'`

2. Can add new keys

- `d['psb26'] = 'Pearl'`

```
d = {'ec1': 'Ezra',  
     'ec2': 'Ezra',  
     'tm55': 'Toni'}
```



Dictionaries are **mutable** (2, after)

1. Can reassign values

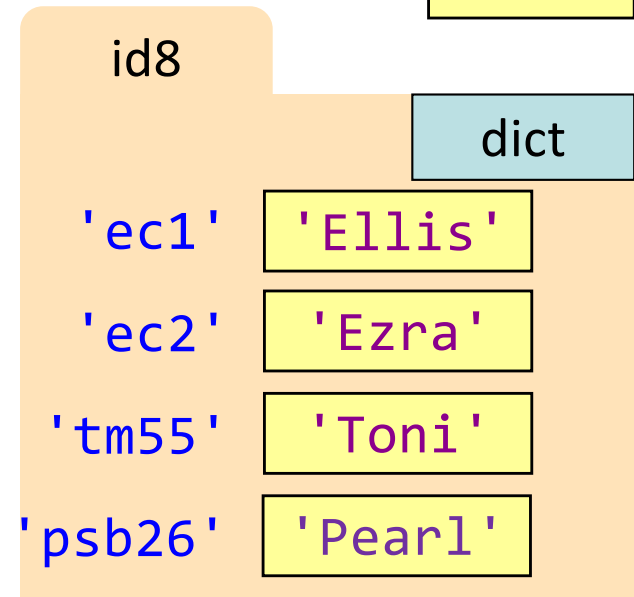
- `d['ec1'] = 'Ellis'`

2. Can add new keys

- `d['psb26'] = 'Pearl'`

```
d = {'ec1': 'Ezra',  
     'ec2': 'Ezra',  
     'tm55': 'Toni'}
```

d id8



Dictionaries are **mutable** (3, before)

1. Can reassign values

- `d['ec1'] = 'Ellis'`

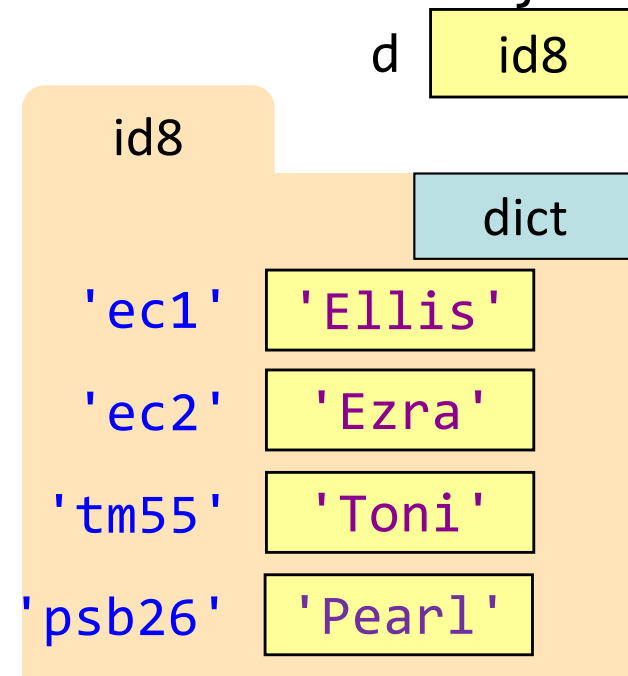
2. Can add new keys

- `d['psb26'] = 'Pearl'`

3. Can delete keys

- `del d['tm55']`

```
d = {'ec1': 'Ezra',  
     'ec2': 'Ezra',  
     'tm55': 'Toni'}
```



Dictionaries are **mutable** (3, after)

1. Can reassign values

- `d['ec1'] = 'Ellis'`

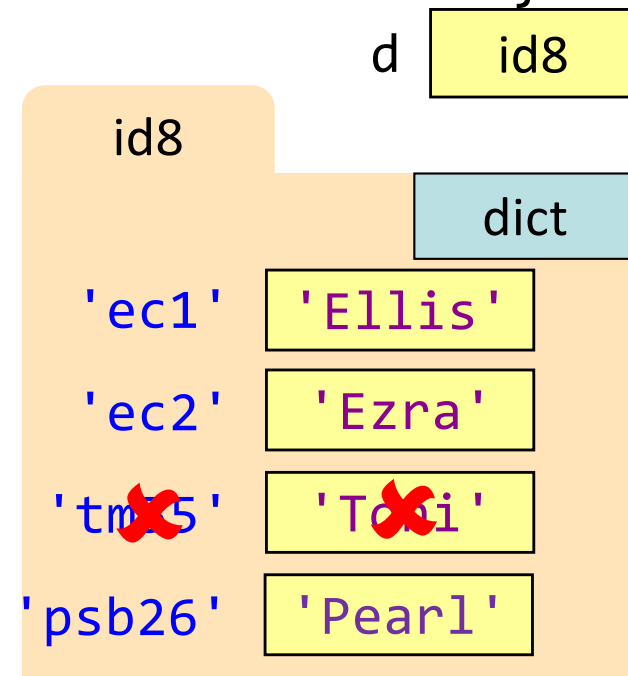
2. Can add new keys

- `d['psb26'] = 'Pearl'`

3. Can delete keys

- `del d['tm55']`

```
d = {'ec1': 'Ezra',  
     'ec2': 'Ezra',  
     'tm55': 'Toni'}
```



Be sure to read
Textbook 11.1-11.5 for
additional examples!

Deleting key deletes both
key and value

A student asked about iterating through a dict

```
students = {'ec1': 'Ezra', 'ec2': 'Ezra',  
           'tm55': 'Toni'}
```

loop variable iterates through each key

```
for netID in students:
```

```
    print(netID + ":" + students[netID])
```

prints to the screen:

```
ec1:Ezra  
ec2:Ezra  
tm55:Toni
```

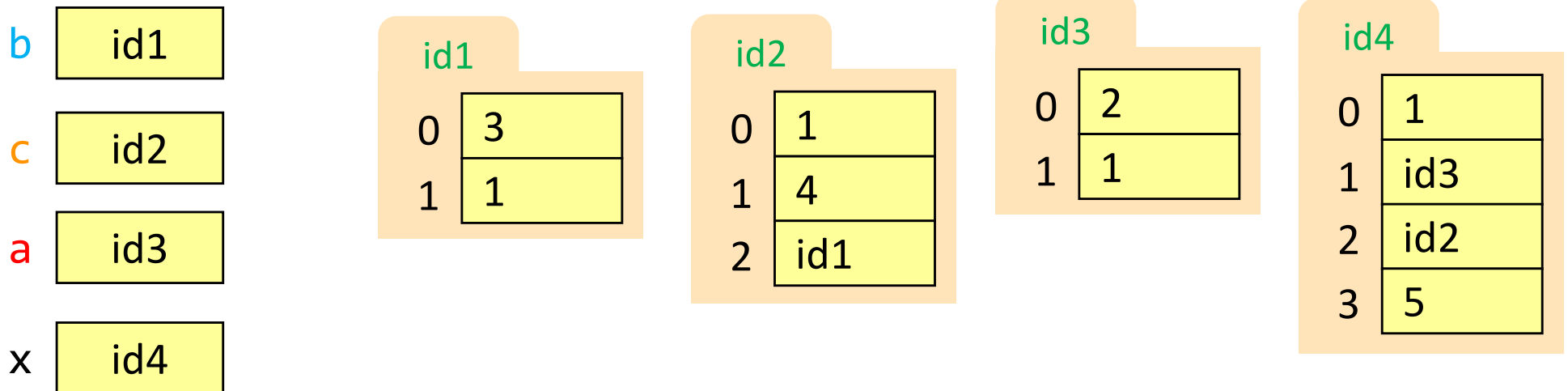
Nested Lists

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

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

Global Space

Heap



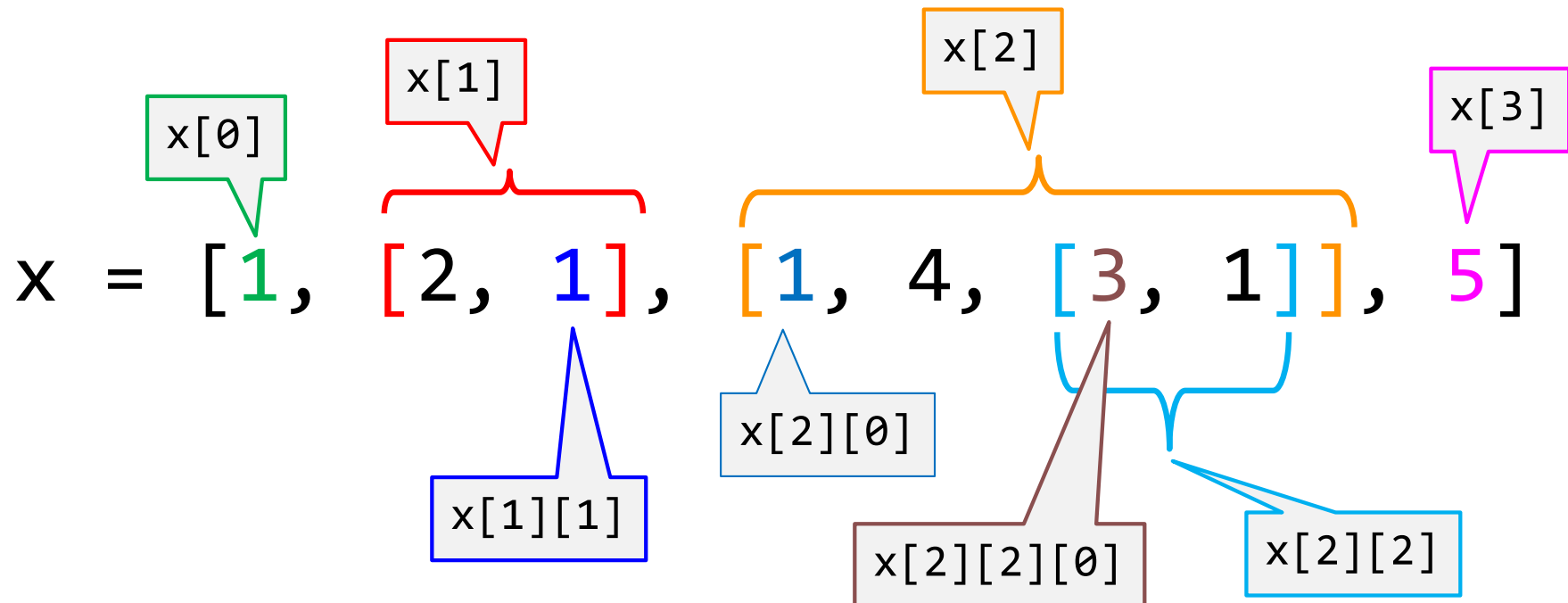
This is drawing accurate, but a little hard to reason about...

Nested Lists

Conceptually, you can visualize nested lists like this:

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

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



"Table-shaped" Two-Dimensional Lists

Table of Data

	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

E.g., lab number

E.g., student ID

Each row, column of the table stores data (a value). Here, the score of the student with ID 1 on lab 3

"Table-shaped" Two-Dimensional Lists

Table of Data

	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

E.g., product ID

E.g., shop ID

Each row, column of the table stores data (a value). Here, the number of units of product 3 sold by the shop with ID 1

"Table-shaped" Two-Dimensional Lists

Table of Data

	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

Column index

Row index

Each row, col has a value

Really a **list of lists**, but convenient to think about it as a **table**, since all inner lists (rows) have the same number of elements.

Store them as a list of lists ("**row-major order**")

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

Overview of Two-Dimensional Lists

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

```
>>> tdlist = [[5,4,7,3],[4,8,9,7],[5,1,2,3],[4,1,2,9]]
```

```
>>> tdlist[3][2] Access value at row 3, col 2
```

```
2
```

```
>>> tdlist[3][2] = 8 Assign value at row 3, col 2
```

```
>>> tdlist
```

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

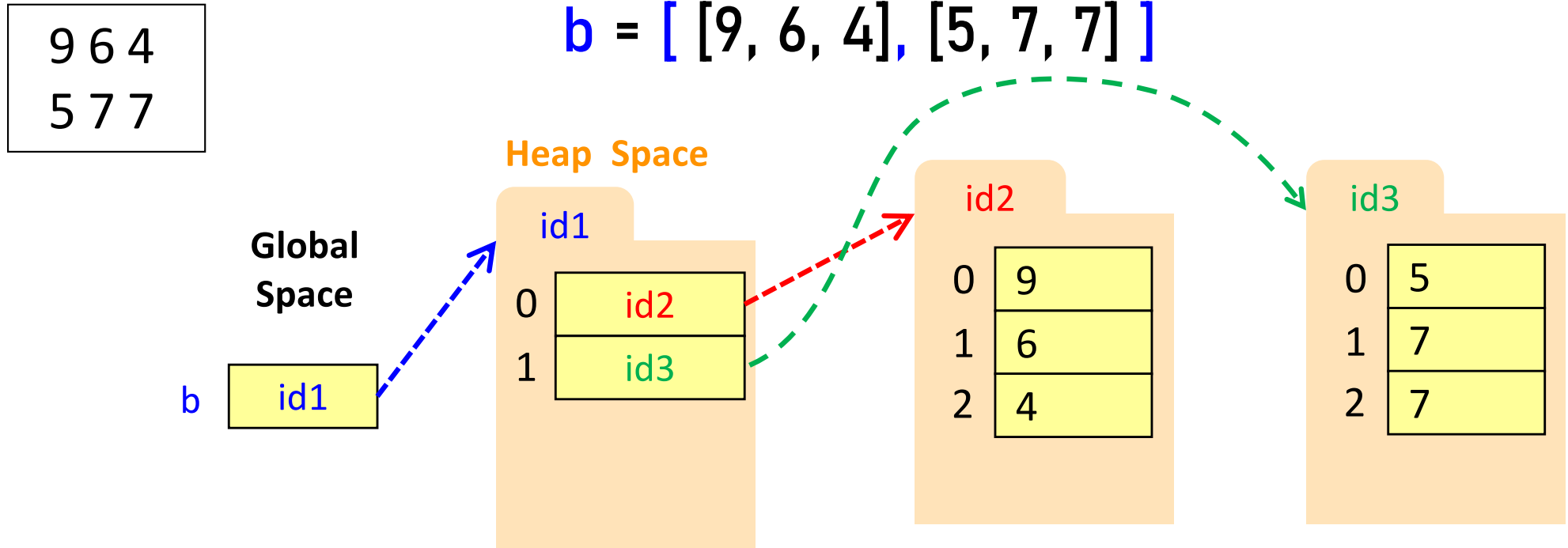
```
>>> len(tdlist) Number of rows of tdlist
```

```
4
```

```
>>> len(tdlist[2]) Number of cols in row 2 of tdlist
```

```
4
```

How Multidimensional Lists are Stored

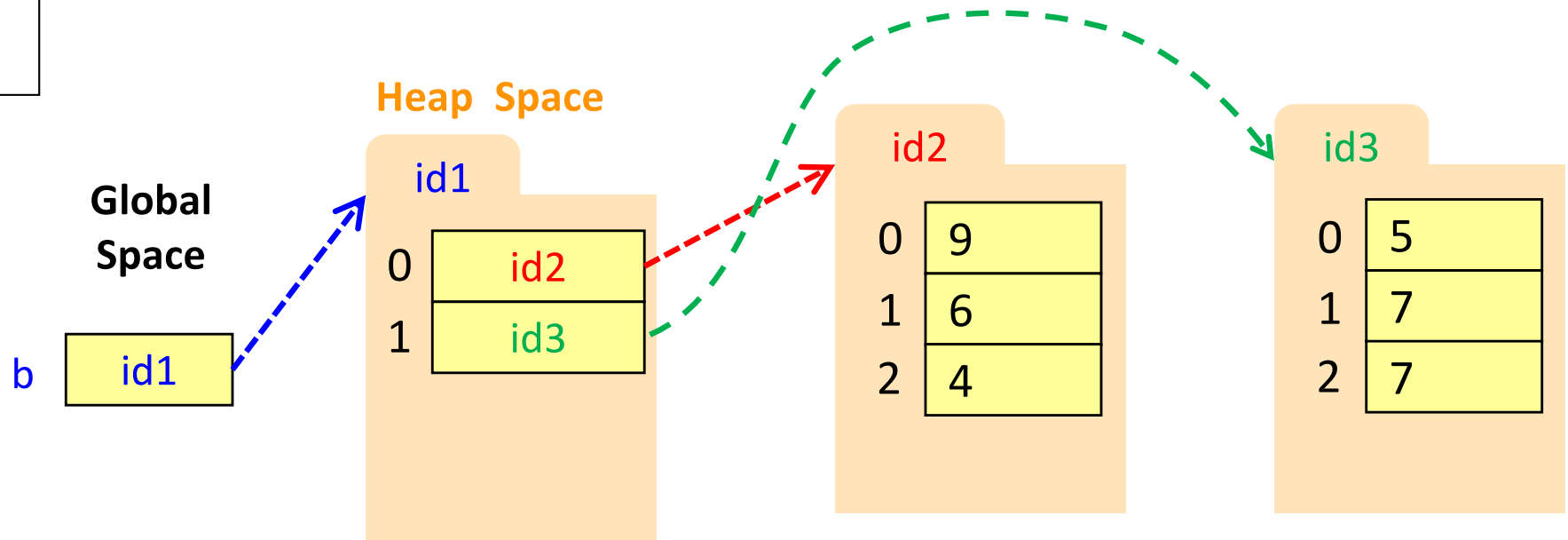


- `b` holds **id** of a one-dimensional list
 - Has `len(b)` elements
- `b[i]` holds **id** of a list
 - Has `len(b[i])` elements

How to access every element of nested list?

9	6	4
5	7	7

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



- `b` holds **id** of a one-dimensional list
 - Has `len(b)` elements # `len(b)` rows
- `b[i]` holds **id** of a list
 - Has `len(b[i])` elements
 - # row `i` has `len(b(i))` elements

Need: a loop to go row by row. At each row, a loop to go column by column.
→ Nested loops!

Exercise 1

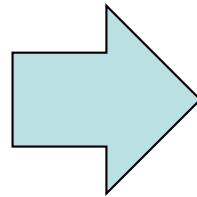
```
def print_all_rows(my_table):  
    """Prints all rows of the table,  
    one row (list) on each line.  
    Preconditions: my_table is a table of numbers  
                  my_table is not empty """
```

Exercise 2

```
def print_all_elements(my_table):  
    """Prints all elements of the table,  
    one element on each line.  
    Preconditions: my_table is a table of numbers  
                  my_table is not empty """
```

Data Wrangling: Transpose Idea

1	2
3	4
5	6
7	8



1	3	5	7
2	4	6	8

2 lists: 4 elements in each

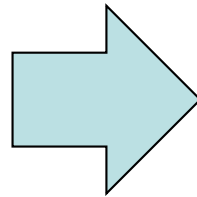
4 lists: 2 elements in each

How to transpose?

- 1st element of each list gets appended to 1st list
- 2nd element of each list gets appended to 2nd list

Data Wrangling: Transpose Idea

1	2
3	4
5	6
7	8



1	3	5	7
2	4	6	8

2 lists: 4 elements in each

4 lists: 2 elements in each

How to transpose?

- 1st element of each list gets appended to 1st list
- 2nd element of each list gets appended to 2nd list

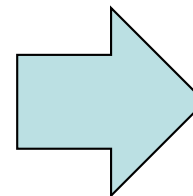
Data Wrangling: Transpose Code

```
def transpose(table):  
    """Returns: copy of table with rows and columns swapped  
    Precondition: table is a (non-ragged) 2d List"""  
    n_rows = len(table)  
    n_cols = len(table[0]) # All rows have same no. cols  
    new_table = [] # Result accumulator
```

```
    return new_table
```

```
d = [[1,2],[3,4],[5,6]]  
d_v2 = transpose(d)
```

1	2
3	4
5	6



1	3	5
2	4	6

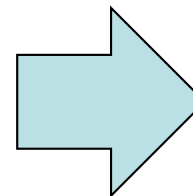
Data Wrangling: Transpose Code

```
def transpose(table):  
    """Returns: copy of table with rows and columns swapped  
    Precondition: table is a (non-ragged) 2d List"""  
    n_rows = len(table)  
    n_cols = len(table[0]) # All rows have same no. cols  
    new_table = [] # Result accumulator
```

```
    return new_table
```

```
d = [[1,2],[3,4],[5,6]]  
d_v2 = transpose(d)
```

1	2
3	4
5	6



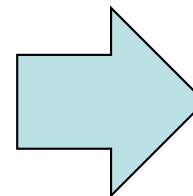
1	3	5
2	4	6

Data Wrangling: Transpose Code

```
def transpose(table):  
    """Returns: copy of table with rows and columns swapped  
    Precondition: table is a (non-ragged) 2d List"""  
    n_rows = len(table)  
    n_cols = len(table[0]) # All rows have same no. cols  
    new_table = [] # Result accumulator  
  
    for c in range(n_cols):  
        row = [] # Single row accumulator  
        for r in range(n_rows):  
            row.append(table[r][c]) # Build up new row  
        new_table.append(row) # Add new row to new table  
    return new_table
```

```
d = [[1,2],[3,4],[5,6]]  
d_v2 = transpose(d)
```

1	2
3	4
5	6

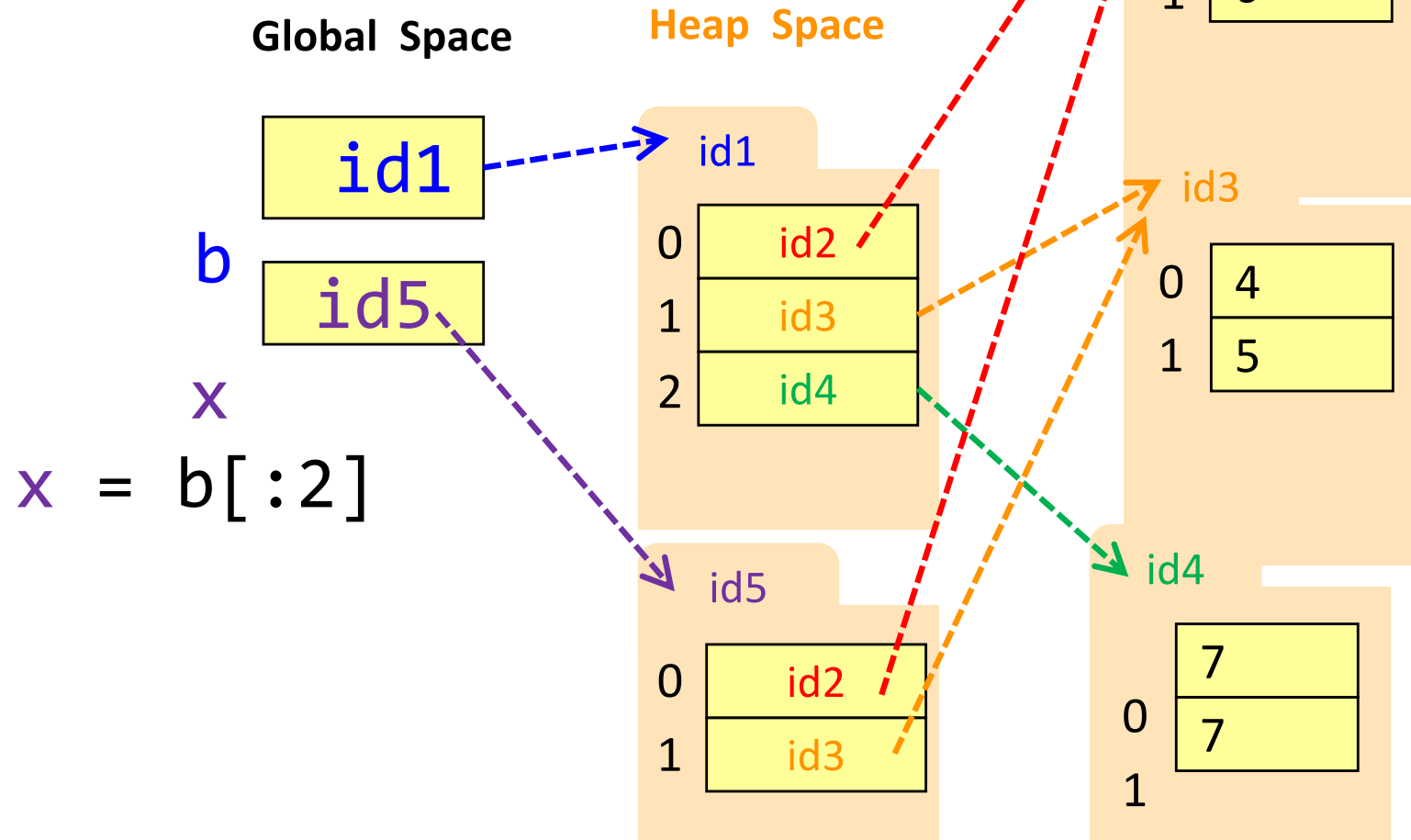


1	3	5
2	4	6

Slices and Multidimensional Lists

- Only “top-level” list is copied.
- Contents of the list are not altered

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



Slices & Multidimensional Lists (Q1)

- 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)
```

What is now in `x`?

A: `[[9,6,10]]`

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 & Multidimensional Lists (Q2)

- 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 is now 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