DICTIONARIES

LECTURE 05–1

JIM FIX, REED COLLEGE CSCI 121

TODAY

- We'll start lecture with ***some group exercises*** on lists.
- We'll look at a second data structure: a ***dictionary***.
- We'll end lecture with ***a short quiz***.
  - It will ask you to write a function that will likely require
    - `if` or `if-else` statements
    - `while` loops

BEFORE WE BEGIN... ANY QUESTIONS?

- List homework?
- Project 1?

SOME GROUP WORK

- Given a list of integers, split it into evens and odds.

```
def evensOdds(xs):
    ????
```

- It should remove the odds from the list it is given, leaving only the evens.
- It should also return a list of the odds.
- Example use:

```
>>> somelist = [1,2,17,111,8]
>>> evensOdds(somelist)
[1, 17, 111]
>>> somelist
[2, 8]
>>>
```

PYTHON LIST SUMMARY

- List creation via enumeration, concatenation, repetition, slicing:

```
[3,1,7]  []  [1,2]+[3,4,5]  [1,2]*4  xs[3:5]  xs[3:]  xs[:]
```

- Accessing contents by index; list length:

```
xs[3]  xs[-1]  len(xs)
```

- Updating contents by indexed assignment:

```
xs[3] = 5
```

- Modifying/mutating a list object:

```
xs.append(5)    xs.extend([8,9,10])    xs.insert(2,357)
xs.pop()        del xs[6]
```

- Checking membership, content equality, object identity:

```
3 in xs      xs == [1,2,3]     xs is ys
```

- Scan according to index using a **`while`** loop:

```
i = 0
while i < len(xs):
    print(xs[i])
    i = i + 1
```

GROUP WORK SOLUTION

- Given a list of integers, this splits it into evens and odds.

```
def evensOdds(xs):
    odds = []
    i = 0
    while i < len(xs):
        if xs[i] % 2 == 1:
            odds.append(xs[i])
            del xs[i]
        else:
            i = i + 1
    return odds
```

MORE GROUP WORK

- Let's work to make a list of slices:

```
def all_prefixes(xs):
    …
```

- Here it is in use:

```
>>> xs = [1,10,100, 1000]
>>> all_prefixes(xs)
[[], [1], [1, 10], [1, 10, 100], [1, 10, 100, 1000]]
```

PYTHON LIST SUMMARY

- List creation via enumeration, concatenation, repetition, slicing:

```
[3,1,7]  []  [1,2]+[3,4,5]  [1,2]*4  xs[3:5]  xs[3:]  xs[:]
```

- Accessing contents by index; list length:

```
xs[3]  xs[-1]  len(xs)
```

- Updating contents by indexed assignment:

```
xs[3] = 5
```

- Modifying/mutating a list object:

```
xs.append(5)    xs.extend([8,9,10])    xs.insert(2,357)
xs.pop()        del xs[6]
```

- Checking membership, content equality, object identity:

```
3 in xs      xs == [1,2,3]     xs is ys
```

- Scan according to index using a **`while`** loop:

```
i = 0
while i < len(xs):
    print(xs[i])
    i = i + 1
```

ALL PREFIXES SOLUTION

- Here is a sample solution:

```
def all_prefixes(xs):
    pfxs = []
    i = 0
    while i <= len(xs):
        pfxs.append(xs[:i])
        i += 1
    return pfxs
```

- Here it is in use:

```
>>> xs = [1,10,100, 1000]
>>> all_prefixes(xs)
[[], [1], [1, 10], [1, 10, 100], [1, 10, 100, 1000]]
```

LISTS OF LISTS

- Lists can be stored within other lists.

```
>>> lls = [[45,19],[8],[17,100,6],[]]
>>> lls[2]
[17, 100, 6]
>>> lls[2][0]
17
>>> lls[2][0] = 7777
>>> lls
[[45, 19], [8], [7777, 100, 6], []]
>>> lls[0].pop()
19
>>> lls[0].extend([0,0,0])
>>> lls
[[45, 19, 0, 0, 0], [8], [7777, 100, 6], []]
>>> lls.append([5,4,3,2])
>>> lls
[[45, 19, 0, 0, 0], [8], [7777, 100, 6], [], [5, 4, 3, 2]]
```

MATRIX REPRESENTATION

- A list of lists is often used to represent a matrix (or a table) of values:

```
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
```

- Here is how we might store this data:

```
>>> table = [[0, 1, 2, 3, 4],[5, 6, 7, 8, 9], [10,11,12,13,14]]
>>>
```

MATRIX REPRESENTATION

- A list of lists is often used to represent a matrix (or a table) of values:

```
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
```

- Here is how we might store this data:

```
>>> table = [[0, 1, 2, 3, 4],[5, 6, 7, 8, 9], [10,11,12,13,14]]
>>> print(????)
```

MATRIX REPRESENTATION

- A list of lists is often used to represent a matrix (or a table) of values:

```
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
```

- Here is how we might store this data:

```
>>> table = [[0, 1, 2, 3, 4],[5, 6, 7, 8, 9], [10,11,12,13,14]]
>>> print(table[2][3])
13
>>>
```

MATRIX REPRESENTATION

- A list of lists is often used to represent a matrix (or a table) of values:

```
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
```

- Here is how we might store this data:

```
>>> table = [[0, 1, 2, 3, 4],[5, 6, 7, 8, 9], [10,11,12,13,14]]
>>> print(table[2][3])
13
>>> table[??][??] = 77
```

MATRIX REPRESENTATION

- A list of lists is often used to represent a matrix (or a table) of values:

```
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
```

- Here is how we might store this data:

```
>>> table = [[0, 1, 2, 3, 4],[5, 6, 7, 8, 9], [10,11,12,13,14]]
>>> print(table[2][3])
13
>>> table[0][1] = 77
```

MATRIX REPRESENTATION

- A list of lists is often used to represent a matrix (or a table) of values:

```
 0 77  2  3  4
 5  6  7  8  9
10 11 12 13 14
```

- Here is how we might store this data:

```
>>> table = [[0, 1, 2, 3, 4],[5, 6, 7, 8, 9], [10,11,12,13,14]]
>>> print(table[2][3])
13
>>> table[0][1] = 77
```

OUR SECOND DATA STRUCTURE: PYTHON DICTIONARIES

OUR SECOND DATA STRUCTURE: PYTHON DICTIONARIES

- Python lets you store a collection of *associations*

```
>>> d = {"bob":35, "mel":24, "betty":29}
>>>
```

OUR SECOND DATA STRUCTURE: PYTHON DICTIONARIES

- Python lets you store a collection of *associations*

```
>>> d = {"bob":35, "mel":24, "betty":29}
>>> d
```

OUR SECOND DATA STRUCTURE: PYTHON DICTIONARIES

- Python lets you store a collection of *associations*

```
>>> d = {"bob":35, "mel":24, "betty":29}
>>> d
{'bob': 35, 'mel': 24, 'betty': 29}
>>>
```

OUR SECOND DATA STRUCTURE: PYTHON DICTIONARIES

- Python lets you store a collection of *associations*

```
>>> d = {"bob":35, "mel":24, "betty":29}
>>> d
{'bob': 35, 'mel': 24, 'betty': 29}
>>> d['bob']
```

OUR SECOND DATA STRUCTURE: PYTHON DICTIONARIES

- Python lets you store a collection of *associations*

```
>>> d = {"bob":35, "mel":24, "betty":29}
>>> d
{'bob': 35, 'mel': 24, 'betty': 29}
>>> d['bob']
35
>>>
```

OUR SECOND DATA STRUCTURE: PYTHON DICTIONARIES

- Python lets you store a collection of *associations*

```
>>> d = {"bob":35, "mel":24, "betty":29}
>>> d
{'bob': 35, 'mel': 24, 'betty': 29}
>>> d['bob']
35
>>> d['mel']
```

OUR SECOND DATA STRUCTURE: PYTHON DICTIONARIES

- Python lets you store a collection of *associations*

```
>>> d = {"bob":35, "mel":24, "betty":29}
>>> d
{'bob': 35, 'mel': 24, 'betty': 29}
>>> d['bob']
35
>>> d['mel']
24
>>>
```

OUR SECOND DATA STRUCTURE: PYTHON DICTIONARIES

- Python lets you store a collection of *associations*

```
>>> d = {"bob":35, "mel":24, "betty":29}
>>> d
{'bob': 35, 'mel': 24, 'betty': 29}
>>> d['bob']
35
>>> d['mel']
24
>>>
```

- This is a data structure called a Python ***dictionary***.
  - A dictionary contains a collection of ***entries***.
  - The left part of each entry is called its ***key***.
  - The right part is that key's ***associated value***.
  - There is *at most one entry* for a key.

OUR SECOND DATA STRUCTURE: PYTHON DICTIONARIES

- Python lets you store a collection of *associations*

```
>>> d = {"bob":35, "mel":24, "betty":29}
>>> d
{'bob': 35, 'mel': 24, 'betty': 29}
>>> d['bob']
35
>>> d['mel']
24
>>> type(d)
<class 'dict'>
>>>
```

- This is a data structure called a Python ***dictionary***.
  - A dictionary contains a collection of ***entries***.
  - The left part of each entry is called its ***key***.
  - The right part is that key's ***associated value***.
  - There is *at most one entry* for a key.

MODIFYING A DICTIONARY'S CONTENTS

- A Python dictionary is also a ***mutable*** data structure.
  - ➡ You can add new key-value pairs, or modify the associated value to a key.
  - ➡ The syntax for adding a new entry and updating an existing entry is the same

```
>>> d = {"bob":35, "mel":24, "betty":29}
>>> d
{'bob': 35, 'mel': 24, 'betty': 29}
>>> d['mel']
24
>>> d['mel'] = 25
>>> d['mel']
25
>>> d
{'bob': 35, 'mel': 25, 'betty': 29}
>>> d['lou'] = 87
>>> d
{'bob': 35, 'mel': 24, 'betty': 29, 'lou': 87}
```

DICTIONARY CONTENT CHECKS

```
>>> d = {"bob":35, "mel":24, "betty":29, "lou": 87}
>>> 'mel' in d      # Does the dictionary contain a key?
True
>>> 'jim' in d
False
>>> 35 in d
False
>>> e =  {"lou": 87,"mel":24, "betty":29, "bob":35}
>>> e == d          # Are the dictionary's contents the same?
True
>>> e is d          # Are they the same object?
False
>>> len(d)          # Get the number of entries.
4
```

BUILDING AND MODIFYING A DICTIONARY

```
>>> d = {}
>>> d['bob'] = 35
>>> d['betty'] = 29
>>> d['mel'] = 24
>>> d
{'bob': 35, 'mel': 24, 'betty': 29}
>>> del d['betty']
>>> d
{'bob': 35, 'mel': 24}
```

LOOPING

```
>>> d = {}
>>> d = {“bob”:35, “betty”:29, “mel”:24}
>>> for k in d:
...     print(k + ” -> ” + str(d[k]))
...
bob -> 35
betty -> 29
mel -> 24
>>>
```

- A `for` loop runs through the ***keys*** of the dictionary.
  - ➡ You can then look up the associated value.

DICTIONARY PRINTING PROCEDURE

- This procedure outputs the contents of a dictionary, one entry per line.

```
def output_entries(d):
    for k in d:
        print(str(k) + " -> " + str(d[k]))
```

DICTIONARY "REVERSE" LOOKUP

- This procedure outputs all the keys that have the same associated value:

```
def output_keys_with(v,d):
    some = False
    for k in d:
        if d[k] == v:
            print("The key " + str(k), end='')
            print(" has value " + str(v) + ".")
            some = True
    if not some:
        print("There were none with", end='')
        print(" value " + str(v) + ".")
```

PYTHON DICTIONARY SUMMARY

- List creation via enumeration of some associations:

```
{'a':89,'b':4}      {}
```

- Accessing contents by key; dictionary size:

```
d['a']                  len(d)
```

- Updating an entry's associated value with key re-assignment:

```
d['a'] = 88
```

- Modifying/mutating a dictionary to add/remove entries:

```
d['c'] = 111
del d['b']
```

- Checking key inclusion, content equality, object identity:

```
'a' in d        d == {'e':78}       d1 is d2
```

- Loop through the keys using a `for` loop.

LIST-LIKE OBJECTS: PYTHON "TUPLES"

- Python lets you make pairs, triples, etc.

```
>>> location = (3.0, 5.0)
>>> location[0]
3.0
>>> location[1]
5.0
>>> type(location)
<class 'tuple'>
>>> location[1] = 6.0
error!
```

- This is a data structure called a Python ***tuple***.
  - ➡ It contains several components.
  - ➡ Each component is accessible by index.
  - ➡ *They cannot be modified!*
    - ✦ They are *immutable*.

LIST-LIKE OBJECTS: PYTHON "TUPLES"

- Python lets you make pairs, triples, etc.

```
>>> student = (("Balob", "Pat"), 3, "Computer Science")
>>> student[0]
("Balob", "Pat")
>>> student[0][1]
"Pat"
>>> student[1]
3
>>> student[2]
"Computer Science"
>>> student[2] = "Studio Art"
error!
```

TUPLES RETURNED BY FUNCTIONS

- Useful for functions that want to return several pieces of info:

```
>>> divmod(137,10)
(13, 7)
>>> q, r = divmod(137, 10)
>>> q
13
>>> r
7
>>>
```

RETURNING A PAIR

- This computes both the minimum and the maximum values of a list:

```
def min_max(xs):
    min = xs[0]
    max = xs[0]
    for x in xs[1:]:
        if x < min:
            min = x
        if x > max:
            max = x
    return (min, max)
```

- Accessing the result:

```
>>> mylist = [100, 27, 89, 137, -10, -23, 67]
>>> min_max(mylist)
(-23, 137)
>>> smallest, largest = min_max(mylist)
>>> smallest
-23
>>> largest
137
>>>
```

TUPLE NOTATION; TUPLE ASSIGNMENT

- Parentheses aren't needed:

```
>>> 13, 7
(13, 7)
>>> first, second = divmod(137, 10)
>>> first
13
>>> second
7
>>> second, first = first, second
>>> first
7
>>> second
13
```

- Programmers will use the tuple notation to swap variables' contents.

OTHER USES

- There are Python libraries and syntax that work naturally with tuples:

```
>>> firsts = ['bob', 'abe', 'dog']
>>> seconds = [19, 22, 7]
>>> list(zip(firsts,seconds))
[('bob', 19), ('abe', 22), ('dog', 7)]
>>> dict(zip(firsts,seconds))
{'bob': 19, 'abe': 22, 'dog': 7}
>>> for f,s in zip(firsts, seconds):
...     print(f, s)
...
bob 19
abe 22
dog 7
>>>
```

QUIZ