Collections

Michael Ernst

CSE 190p

University of Washington

Needed for Homework 4 (social networking assignment)

- Collections: lists, sets, dictionaries
- Sorting
- Graphs

Outline for today

- Collections (built-in data structures)
  - Lists
    - Sorting
  - Sets
    - Order independence
  - Dictionaries (mappings)
- Graphs

How to evaluate list expressions

There are two new forms of expression:

- [a, b, c, d]   list creation
  - To evaluate:
    - evaluate each element to a value, from left to right
    - make a list of the values
  - The elements can be arbitrary values, including lists
    - ["a", 3, 3.14*r*r, fahr_to_cent(-40), [3+4, 5*6]]

List expression

- a[b]   list indexing or dereferencing

Index expression

  - To evaluate:
    - evaluate the list expression to a value
    - evaluate the index part to a value
    - if the list value is not a list, execution terminates with an error
    - if the element is not in range (not a valid index), execution terminates with an error
    - the value is the given element of the list value

List slicing

`mylist[startindex : endindex]` evaluates to a sublist of the original list

- **`mylist[index]`** evaluates to an element of the original list

- Arguments are like those to the **`range`** function
  - start index is inclusive
  - end index is exclusive
  - optional step argument: **`mylist[st : end : step]`**
- See handout for practice problems

Sorting

```
hamlet = "to be or not to be that is the
question whether tis nobler in the mind to
suffer".split()

print "hamlet:", hamlet

print "sorted(hamlet):", sorted(hamlet)
print "hamlet:", hamlet

print "hamlet.sort():", hamlet.sort()
print "hamlet:", hamlet
```

- Lists are mutable – they can be changed
  - including by functions

Customizing the sort order

Goal: sort a list of names *by last name*

```
names = ["Isaac Newton", "Albert Einstein", "Niels Bohr", "Charles
Darwin", "Louis Pasteur", "Sigmund Freud", "Galileo Galilei"]
```

```
print "names:", names
```

This does not work:

```
print "sorted(names):", sorted(names)
```

When sorting, how should we compare these names?

```
"Niels Bohr"
"Charles Darwin"
```

A sort key is a different value that you use to sort a list, instead of the actual values in the list

```
def last_name(str):
    return str.split(" ")[1]
```

```
print 'last_name("Isaac Newton"):', last_name("Isaac Newton")
```

Two ways to use a sort key

1. Create a different list that contains the sort key, sort it, then extract the part you care about

```
keyed_names = [[last_name(name), name] for name in names]
print "keyed_names:", keyed_names

print "sorted(keyed_names):", sorted(keyed_names)

print "sorted(keyed_names, reverse = True):"
print sorted(keyed_names, reverse = True)
```

(This works because Python compares two elements that are lists *elementwise*.)

```
sorted_names = [keyed_name[1] for keyed_name in sorted(keyed_names, reverse = True)]
print "sorted_names:", sorted_names
```

2. Supply the **`key`** argument to the **`sorted`** function or the **`sort`** function

```
print "sorted(names, key = last_name):"
print sorted(names, key = last_name)

print "sorted(names, key = last_name, reverse = True):"
print sorted(names, key = last_name, reverse = True)
```

Sets

- Mathematical set: a collection of values, without duplicates or order
- Two ways to create a set:
  - Direct mathematical syntax

    ```
    odd = { 1, 3, 5 }
    prime = { 2, 3, 5 }
    ```

    Cannot express empty set: "**{ }**" means something else ☹
  - Construct from a list

    ```
    odd = set([1, 3, 5])
    prime = set([2, 3, 5])
    empty = set([])
    ```

    Python always prints using this syntax
- Order does not matter

  { 1, 2, 3 } == { 3, 2, 1 }
- No duplicates

  set([3,1,4,1,5]) == { 5, 4, 3, 1 }

Set operations

```
odd = { 1, 3, 5 }
prime = { 2, 3, 5 }
```

- union $\cup$ Python: `|` `odd | prime` $\Rightarrow$ { 1, 2, 3, 5 }
- intersection $\cap$ Python: `&` `odd & prime` $\Rightarrow$ { 3, 5 }
- difference \ or - Python: `-` `odd – prime` $\Rightarrow$ { 1 }
- membership $\in$ Python: `in` `4 in prime` $\Rightarrow$ False
- Iteration over sets:

```
# iterates over items in arbitrary order
for item in myset:
    …
```

- Add one element to a set:

```
myset.add(newelt)
myset = myset | {newelt}
```

- Think in terms of set operations,
  not in terms of iteration and element operations
  - Shorter, clearer, less error-prone, faster

Practice with sets

Not every value may be placed in a set

- Set elements must be immutable values
  - int, float, bool, string, *tuple*
  - *not*: list, set, dictionary
- Goals
  - after "**`myset.add(x)`**", **`x in myset`** ⇒ True
  - **`y in myset`** always evaluates to the same value

  Both conditions should hold until **`myset`** is changed
- Mutable elements can violate these goals

```
list1 = ["a", "b"]
list2 = list1
list3 = ["a", "b"]
myset = {list1}                ⇐ Hypothetical; actually illegal in Python
list1 in myset  ⇒ True
list3 in myset  ⇒ True
list2.append("c")
list1 in myset  ⇒ ???
list3 in myset  ⇒ ???
```

Computing a histogram

- Recall exercise from previous lecture:

  For each word in a text, record the number of times that word appears
- *Without* thinking about any Python data structures, how would you solve this?
  - Always start by thinking about the data, *not* by thinking about how you would implement it

```
hamlet = "to be or not to be that is
the question whether tis nobler in the
mind to suffer".split()
```

Dictionaries or mappings

- A dictionary maps each *key* to a *value*

```
d = { }
us_wars1 = {
    "Revolutionary" : [1775, 1783],
    "Mexican" : [1846, 1848],
    "Civil" : [1861, 1865] }
us_wars2 = {
    1783: "Revolutionary",
    1848: "Mexican",
    1865: "Civil" }
```

- Syntax just like arrays, for accessing and setting:

```
us_wars2[1783][1:10]
```

$\Rightarrow$ "evolution"

```
us_wars1["WWI"] = [1917, 1918]
```

- Order does not matter

  { 5 : 25, 6 : 36, 7 : 49 } == { 7 : 49, 5: 25, 6 : 36 }

Not every value is allowed to be a key

- Keys must be immutable values
  - int, float, bool, string, *tuple*
  - *not*: list, set, dictionary
- Goals
  - after "**`mydict[x] = y`**", **`mydict[x]`** ⇒ y
  - if **`a == b`**, then **`mydict[a] == mydict[b]`**

  These conditions should hold until **`mydict`** is changed
- Mutable keys can violate these goals

```
list1 = ["a", "b"]
list2 = list1
list3 = ["a", "b"]
mydict = {}
mydict[list1] = "z"            ⇐ Hypothetical; actually illegal in Python
mydict[list3] ⇒ "z"
list2.append("c")
mydict[list1] ⇒ ???
mydict[list3] ⇒ ???
```

Graphs

- A graph can be thought of as:
  - a collection of edges
  - for each node, a collection of neighbors