DICTIONARIES AND MUTATION 4

COMPUTER SCIENCE 61A

October 2, 2015

1 Mutable Lists

Let's imagine you order a mushroom and cheese pizza from Domino's, and that they represent your order as a list:

```
>>> pizza1 = ['cheese', 'mushrooms']
```

A couple minutes later, you realize that you really want onions on the pizza. Based on what we know so far, Domino's would have to build an entirely new list to add onions:

```
>>> pizza2 = pizza1 + ['onions'] # creates a new python list
>>> pizza2
['cheese', mushrooms', 'onions']
>>> pizza1 # the original list is unmodified
['cheese', 'mushrooms']
```

But this is silly, considering that all Domino's had to do was add onions on top of `pizza1` instead of making an entirely new `pizza2`.

Python actually allows you to *mutate* some objects, includings lists and dictionaries. Mutability means that the object's contents can be changed. So instead of building a new `pizza2`, we can use `pizza1.append('onions')` to mutate `pizza1`.

```
>>> pizza1.append('onions')
>>> pizza1
['cheese', 'mushrooms', 'onions']
```

Although lists and dictionaries are mutable, many other objects, such as numeric types, tuples, and strings, are *immutable*, meaning they cannot be changed once they are created. We can use the familiar indexing operator to mutate a single element in a list. For instance `lst[4]='hello'` would change the fifth element in `lst` to be the string `'hello'`. In

addition to the indexing operator, lists have many mutating methods. List *methods* are functions that are bound to a specific list. Some useful list methods are listed here:

1. `append(el)` adds `el` to the end of the list
2. `insert(i, el)` insert `el` at index `i`
3. `remove(el)` removes the first occurrence of `el` in list, otherwise errors
4. `sort()` sorts elements of list *in place*

List methods are called via *dot notation*, as in:

```
>>> colts = ['andrew luck', 'reggie wayne']
>>> colts.append('trent richardson')
```

None of the mutating list methods *return* a new list — they simply modify the original list and return `None`.

1.1 Questions

1. Consider the following definitions and assignments and determine what Python would output for each of the calls below *if they were evaluated in order*. It may be helpful to draw the box and pointers diagrams to the right in order to keep track of the state.

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
>>> lst1 == lst2 #compares each value


>>> lst1 is lst2 #compares references


>>> lst2 = lst1
>>> lst2 is lst1


>>> lst1.append(4)
>>> lst1


>>> lst2


>>> lst2[1] = 42
>>> lst2


>>> lst1 = lst1 + [5]
```

```
>>> lst1 == lst2

>>> lst1

>>> lst2

>>> lst2 is lst1
```

2. Write a function that removes all instances of an element from a list.

```
def remove_all(el, lst):
    """
    >>> x = [3, 1, 2, 1, 5, 1, 1, 7]
    >>> remove_all(1, x)
    >>> x
    [3, 2, 5, 7]
    """
```

3. Write a function that takes in two values `x` and `el`, and a list, and adds as many `el`'s to the end of the list as there are `x`'s.

```
def add_this_many(x, el, lst):
    """ Adds el to the end of lst the number of times x occurs
    in lst.
    >>> lst = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5]
    """
```

2 Dictionaries

Dictionaries are data structures which map keys to values. Dictionaries in Python are unordered, unlike real-world dictionaries — in other words, key-value pairs are not arranged in the dictionary in any particular order. Let's look at an example:

```
>>> pokemon = {'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['pikachu']
25
>>> pokemon['jolteon'] = 135
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['ditto'] = 25
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148,
'ditto': 25, 'mew': 151}
```

The *keys* of a dictionary can be any *immutable* value, such as numbers, strings, and tuples. Dictionaries themselves are mutable; we can add, remove, and change entries after creation. There is only one value per key, however — if we assign a new value to the same key, it overrides any previous value which might have existed.

To access the value of `dictionary` at `key`, use the syntax
`dictionary[key]`

Element selection and reassignment work similarly to sequences, except the square brackets contain the key, not an index.

1. Predict what Python would output given the following inputs.

```
>>> 'mewtwo' in pokemon
```

```
>>> len(pokemon)
```

```
>>> pokemon['ditto'] = pokemon['jolteon']
>>> pokemon[('diglett', 'diglett', 'diglett')] = 51
>>> pokemon[25] = 'pikachu'
>>> pokemon
```

```
>>> pokemon['mewtwo'] = pokemon['mew'] * 2
>>> pokemon


>>> pokemon[['firetype', 'flying']] = 146
```

Note that the last example demonstrates that dictionaries cannot use other mutable data structures as keys. However, dictionaries can be arbitrarily deep, meaning the *values* of a dictionary can be themselves dictionaries.

- To add `val` corresponding to `key` *or* to replace the current value of `key` with `val`:
  ```
  dictionary[key] = val
  ```
- To iterate over a dictionary's keys:
  ```
  for key in dictionary: #OR for key in dictionary.keys()
      do_stuff()
  ```
- To iterate over a dictionary's values:
  ```
  for value in dictionary.values():
      do_stuff()
  ```
- To iterate over a dictionary's keys and values:
  ```
  for key, value in dictionary.items():
      do_stuff()
  ```
- To remove an entry in a dictionary:
  ```
  del dictionary[key]
  ```
- To get the value corresponding to `key` and remove the entry:
  ```
  dictionary.pop(key)
  ```

2. Given a (non-nested) dictionary `d`, write a function which deletes all occurrences of `x` as a value. You cannot delete items in a dictionary as you are iterating through it.

```
def remove_all(d, x):
    """
    >>> d = {1:2, 2:3, 3:2, 4:3}
    >>> remove_all(d, 2)
    >>> d
    {2: 3, 4: 3}
    """
```

3. Given an arbitrarily deep dictionary `d`, replace all occurences of `x` *as a value (not a key)* with `y`. Hint: You will need to combine iteration and recursion.

```
def replace_all_deep(d, x, y):
    """
    >>> d = {1: {2: 3, 3: 4}, 2: {4: 4, 5: 3}}
    >>> replace_all_deep(d, 3, 1)
    >>> d
    {1: {2: 1, 3: 4}, 2: {4: 4, 5: 1}}
    """
```

3 Extra List Practice!

1. Reverse a list *in place*, i.e. mutate the given list itself, instead of returning a new list.

```
def reverse(lst):
    """ Reverses lst in place.
    >>> x = [3, 2, 4, 5, 1]
    >>> reverse(x)
    >>> x
    [1, 5, 4, 2, 3]
    """
```

2. Write a function that rotates the elements of a list to the right by k. Elements should not "fall off"; they should wrap around the beginning of the list. `rotate` should return a new list. To make a list of n 0's, you can do this: `[0] * n`

```
def rotate(lst, k):
    """ Return a new list, with the same elements
        of lst, rotated to the right k.
    >>> x = [1, 2, 3, 4, 5]
    >>> rotate(x, 3)
    [3, 4, 5, 1, 2]
    """
```