OBJECTS, LISTS, DICTIONARIES, AND MUTABLE DATA 3a

COMPUTER SCIENCE 61A

July 9, 2013

1 Lists

A list is an ordered collection of values. You can have a list of integers, a list of strings, or even a mix of any types of values you want; this means that the list need not be homogenous. You can add and remove items to and from a list them by calling list methods, and you can access elements through the index notation. Let's look at an example:

```
>>> fantasy_team = []
>>> fantasy_team.append("frank gore")
>>> print(fantasy_team)
['frank gore']
>>> fantasy_team.append("calvin johnson")
>>> print(fantasy_team[1])
calvin johnson
>>> fantasy_team.remove("calvin johnson")
>>> fantasy_team[0] = "aaron rodgers"
>>> print(fantasy_team)
['aaron rodgers']
```

Lists can be created using square braces, and likewise, their elements can be accessed via square braces. Just like tuples, lists are zero-indexed. Let's try out some basics.

1.1 Basics

1. What would Python print?

```
>>> a = [1, 5, 4, 2, 3]
>>> print(a[0], a[-1])

>>> a[4] = a[2] + a[-2]
>>> a

>>> len(a)

>>> 4 in a

>>> a[1] = [a[1], a[0]]
>>> a
```

1.2 List methods

In addition to the indexing operator, lists have many mutating methods, some examples of which are listed here:

1. `append(el)` → Adds `el` to the end of the list
2. `index(el)` → Returns the index of `el` if it occurs in the list, otherwise errors.
3. `insert(i, el)` → Insert `el` at index `i`
4. `remove(el)` → Removes the first occurrence of `el` in list, otherwise errors
5. `sort()` → Sorts elements of list *in place*

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

```
>>> fruits = ['apple', 'pineapple']
>>> fruits.append('banana')
```

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

```
def remove_all(el, lst):
    """Removes all instances of el from lst.
    >>> x = [3, 1, 2, 1, 5, 1, 1, 7]
    >>> remove_all(1, x)
    >>> x
    [3, 2, 5, 7]
    """
```

2. Write a function that takes in two values, x and y, and a list, and adds as many y's to the end of the list as there are x's. Do not use the built-in function count.

```
def add_this_many(x, y, lst):
    """ Adds y 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]
    """
```

1.3 Slicing

Like tuples, lists also support slicing notation, allowing you to retrieve multiple elements of a list at once. Slicing a list returns a *new* list. Slicing has the following syntax:

```
lst[start:end:interval]
```

where start, end, and interval are integers. The slice includes the element at start and every interval elements up to but not including the element at end. It is legal to omit one or more of start, end, and incr; they default to 0, `len(lst)`, and 1, respectively. Start and end can be negative, meaning you count from the end.

```
>>> a = [0, 1, 2, 3, 4, 5, 6]
>>> a[1:4]
[1, 2, 3]
>>> a[1:6:2]
[1, 3, 5]
>>> a[:4] # equivalent to a[0:4]
[0, 1, 2, 3]
>>> a[3:] # equivalent to a[3:len(a)]
[3, 4, 5, 6]
>>> a[1:4:] # equivalent to a[1:4:1] or a[1:4]
[1, 2, 3]
>>> a[-1:]
[6]
```

1. What would Python print?

```
>>> a = [3, 1, 4, 2, 5, 3]
>>> a[:4]


>>> a


>>> a[1::2]


>>> a[:]


>>> a[4:2]


>>> a[1:-2]


>>> a[::-1]
```

1.4 For loops

There are two main methods of looping through lists.

- `for el in lst` → loops through the elements in lst
- `for i in range(len(lst))` → loops through the valid, positive indices of lst

If you do not need indices, looping over elements is usually more clear. Let's try this out.

1. In the homework, you reversed rlists iteratively and recursively. Let's reverse Python lists *in place*, meaning mutate the passed in list itself, instead of returning a new list. Why is this solution preferred?

```
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]
    """
```

1.5 Higher order functions

Many times, we wish an operation to be applied to all elements of a list. Python has methods built in to help us with these tasks:

- `map(fn, lst)` → applies fn to each element in lst
- `filter(pred, lst)` → keeps those elements in lst that satisfy the predicate
- `reduce(accum, lst, zero_value)` → repeatedly calls the accumulator, which takes in two arguments and returns a single value, on elements of lst

We can also use higher order functions in *list comprehensions*. List comprehensions are a compact way to apply some operations to a sequence. They look like this:

```
[expression for value in sequence if predicate]
```

where the if clause is optional.

1. What would Python print?

```
>>> l_1, l_2 = lambda x: 3*x + 1, lambda x: x % 2 == 0
>>> list(filter(l_2, map(l_1, [1,2,3,4])))


>>> [x*x - x for x in [1, 2, 3, 4] if x > 2]


>>> [[y*2 for y in [x, x+1]] for x in [1,2,3,4]]
```

2 Dictionaries

Recall that *dictionaries* are data structures that map *keys* to *values*. Dictionaries are usually unordered (unlike real-world dictionaries) – in other words, the key-value pairs are not arranged in the dictionary in any particular order. Let's look at an example:

```
>>> superbowls = {'joe montana': 4, 'tom brady':3, 'joe flacco': 0}
>>> superbowls['tom brady']
3
>>> superbowls['peyton manning'] = 1
>>> superbowls
{'peyton manning': 1, 'tom brady': 3, 'joe flacco': 0, 'joe montana': 4}
>>> superbowls['joe flacco'] = 1
>>> superbowls
{'peyton manning': 1, 'tom brady': 3, 'joe flacco': 1, 'joe montana': 4}
```

Dictionaries are indexed with similar syntax as sequences, only they use keys, which can be any immutable value, not just numbers. Dictionaries themselves are mutable; we can

add, remove, and change entries after creation. There is only one value per key, however, in a dictionary (we call this *injective* or one-to-one).

1. Continuing from above, what would Python print?

```
>>> 'colin kaepernick' in superbowls


>>> len(superbowls)


>>> superbowls['peyton manning'] = superbowls['joe montana']
>>> superbowls[('eli manning', 'giants')] = 2
>>> superbowls[3] = 'cat'
>>> superbowls


>>> superbowls[('eli manning', 'giants)] = \
        superbowls['joe montana'] + superbowls['peyton manning']
>>> superbowls[['steelers', '49ers']] = 11
>>> superbowls
```

Dictionaries in general can be arbitrarily deep, meaning their values can be dictionaries themselves. Let's get practice traversing these deep structures. To do so, we'll need to know a couple more things about dictionaries.
To iterate over a dictionary's keys:

```
for k in d.keys():
    ...
```

and to remove an entry:

```
del dictionary[key]
```

2. Given a dictionary replace all occurrences of x as the value with y.

```
def replace_all(d, x, y):
    """Replaces all values of x with y.
    >>> d = {1: {2:3, 3:4}, 2:{4:4, 5:3}}
    >>> replace_all(d,3,1)
    >>> d
    {1: {2: 1, 3: 4}, 2: {4: 4, 5: 1}}
    """
```

3. Given a (non-nested) dictionary delete all occurences of a value. You cannot delete items in a dictionary as you are iterating through it.

```
def rm(d, x):
    """Removes all pairs with value x.
    >>> d = {1:2, 2:3, 3:2, 4:3}
    >>> rm(d,2)
    >>> d
    {2:3, 4:3}
    """
```