MUTABLE LISTS AND DICTIONARIES 4

COMPUTER SCIENCE 61A

Sept. 24, 2012

1 Lists

Lists are similar to tuples: the order of the data matters, their indices start at 0. The big difference is that lists are *mutable* – lists can be modified after creation, while tuples cannot.

Lists are denoted in Python by square brackets. Indexing lists is exactly the same as with tuples:

```
>>> a = [1, 2, 3, 4]     # creates a 4 element list
>>> a[0]                 # lists are also zero-based
1
>>> a[100]               # can't index out of bounds!
IndexError: list index out of range
```

Mutating a list is where we see a real difference between tuples and lists:

```
>>> lst = [1, 2, 3, 4]
>>> a[1] = 100           # change second element to 100
>>> a
[1, 100, 3, 4]           # changed!

>>> tup = (1, 2, 3, 4)  # what if we tried to change a tuple?
>>> b[1] = 100
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

Lists also have functions that allow you to modify them, including (but not limited to) the following:

- `append(elem)`: add `elem` to the original list
- `extend(seq)`: adds the elements in `seq` to the end of the original list
- `remove(elem)`: removes the first instance of `elem` from the original list
- `pop(i)`: removes the element at index `i` and returns that removed element (if `i` is not given, the last element is removed and returned

NOTE: none of these functions actually create new lists – they all modify the original list.

NOTE: these functions are accessed using "dot notation" – we'll learn later on why that's the case. See the following code:

```
>>> c = ['hi', 'bye']
>>> c.append('bye')
>>> c
['hi', 'bye', 'bye']
```

1.1 What would Python print?

1. Basics:

```
>>> a = [5, 2, 6, 3]
>>> a[1000]
```

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

```
>>> list((1, 2, 3))
```

```
>>> len(a)
```

```
>>> 6 in a
```

```
>>> a[0] = a
>>> a
```

```
>>> a[0]
```

2. List methods

```
>>> lst = [4, 2]
>>> lst.extend([4, 5])
>>> lst
```

```
>>> lst.append([6, 7])
```

```
>>> lst.insert(2, 'hi')
```

```
>>> bob = [4, 2, 3, 1]
>>> sorted(bob)
```

```
>>> bob
```

```
>>> bob.sort()
>>> bob
```

1.2 Writing list functions

For the following questions, the functions

- should mutate the original list
- should NOT create any new lists
- should NOT return anything

Functions that do not create new lists are said to be "in place." Such functions are often desirable because they do not require extra memory to operate.

1. Implement the function `map_mut`, which applies a function `fn` onto every element of a list called `lst`.

```
def map_mut(fn, lst):
    """Maps fn onto lst by mutation.

    >>> lst = [1, 2, 3, 4]
    >>> map_mut(lambda x: x**2, lst)
    >>> lst
    [1, 4, 9, 16]
    """
```

2. Define `shift_left`, a function that takes a list and shifts each element in the list to the left by `n` indices. If elements start "falling off" on the left, they are placed back on the right. *NOTE*: you may assume that `n` is a non-negative integer.

```
def shift_left(lst, n):
    """Shifts the elements of lst over by n indices.

    >>> lst = [1, 2, 3, 4, 5]
    >>> shift_left(lst, 2)
    >>> lst
    [3, 4, 5, 1, 2]
    """
```

3. Define `filter_mut`, which takes a list and filters out elements that don't satifsy a given predicate.

```
def filter_mut(pred, lst):
    """Filters lst by mutating it.

    >>> lst = [1, 2, 3, 4]
    >>> is_even = lambda x: x % 2 == 0
    >>> filter_mut(is_even, lst)
    >>> lst
    [2, 4]
    """
```

1.3 Slicing

Like tuples, lists also support *slicing* notation – you are able to retrieve multiple elements of a list at once. There are two ways to slice a list:

- `lst[start:end]` where `start` and `end` are indices (i.e. integers). The slice includes the element at `start` and all elements up to *but not including* the element at `end`
- `lst[start:end:incr]` where `start`, `end`, and `incr` are integers

It is legal to omit one or more of `start`, `end`, and `incr`.

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

NOTE: slicing does NOT mutate the original list – it creates a new list.

1.4 Questions

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]
```

1.5 List comprehensions

Often times, we will be using `for` loops to iterate over lists. In Python, there is a built in syntax to make this process elegant – it's called *list comprehension*. The general syntax for a list comprehension is the following:

```
[ <map expr> for <name> in <sequence> if <filter expr> ]
```

The **if** `<filter expr>` part is optional.

Consider the following code:

```
def foo(lst):
    new = []
    for elem in lst:
        if elem % 2 == 0:
            new += [elem**2]
    return new
```

There's nothing wrong with this code, but list comprehensions enable us to make it even more concise:

```
def foo(lst):
    return [ elem**2 for elem in lst if elem %2 == 0 ]
```

List comprehensions are designed to read like English. The previous line should be read as "the new items are elem**2 for each elem in the lst, only if each elem is even."

One other thing: list comprehensions create new lists. They do not mutate the original sequence, since the original sequence might not be mutable at all.

1.6 Questions

1. What would Python print?

```
>>> seq = range(5)
>>> x = [x * x for x in seq]
>>> x
```

```
>>> [elem + 1 for elem in (1, 2, 3, 4) if elem % 2 == 0]
```

```
>>> [x + y for x in (1, 4) for y in [5, 2]]
```

```
>>> [[x + y for x in (1, 4)] for y in [5, 2]]
```

2 Dictionaries

Python also has *dictionaries*, a data structure that stores *mappings* of keys to values. If you give a dictionary a *key*, it will give you the associated *value*. For example:

```
>>> d = {'a': 1, 'b': 2, 'c': 1}
>>> d['a']
1
>>> d['b']
2
```

Unlike with lists and tuples, we query dictionaries with *keys* (like 'a'), not with indices (e.g. it doesn't make sense to ask for the "first" item in a dictionary).

Dictionaries have a few special qualities:

- **not ordered**: the order in which you enter key/value may not be the order in which they are stored in the dictionary.
- **mutable**: you can add and remove keys from dictionaries
- **one-to-one**: a key can only map to one value (although that value can be a tuple of elements)

The following code demonstrates how to work with dictionaries:

```
>>> names = {}                    # create an empty dictionary
>>> names
{}
>>> names['john'] = 'denero'      # add key/value pair
>>> names['john']                 # given key, retrieve value
'denero'
>>> names['john'] = 'doe'         # change an existing key/value pair
>>> names['john']
'doe'
>>> del names['john']             # delete a key/value pair
>>> names                         # names is now empty
{}
>>> names['john']                 # john doesn't exist anymore!
KeyError: 'john'
```

One thing to note: **keys must be immutable**. That means `int`s, `float`s, and strings can be used as keys, since they cannot be changed after creation. However, *mutable* objects like lists cannot be used as keys. This makes sense, since a modifiable key would make it impossible to keep track of which key to use.

2.1 Questions

1. Write `make_inverse_dict(d)` that returns a new dictionary with the 'inverse' mapping. The 'inverse' mapping of a dictionary `d` is a new dictionary that maps each of `d`'s values to all keys in `d` that mapped to it. For instance,

```
>>> d1 = {'call': 3, 'me': 2, 'maybe': 3}
>>> d2 = make_inverse_dict(d1)
>>> d2 #note that we know nothing about the order of dictionaries
{3: ('maybe', 'call'), 2: ('me',)}
```

   The ordering of the tuple of keys doesn't matter, i.e., `d2` could have instead been `3: ('call', 'maybe'), 2: ('me',)`.

```
def make_inverse_dict(d):
```