MUTABLE LISTS AND DICTIONARIES 8

COMPUTER SCIENCE 61A

July 12, 2012

1 Lists

The Python list is a data structure very similar to a tuple: the primary difference is that lists are mutable and tuples are not. Mutability lets us create a data structure that we can update on the go, rather than having to recreate the data structure every time we want to make a change, as we did with tuples.

Constructing a list is very similar to constructing tuples, except that instead of parentheses, we use square brackets.

```
>>> empty_list = list()
>>> empty_list
[]
>>> x = [3, 4, 5, 'hello']   #constructs a 4 element list
>>> x[2]            #selects the 2nd index
5
```

Mutation occurs when we perform assignment after selecting an index:

```
>>> x[2] = 'CHANGED'
>>> x
[3, 4, 'CHANGED', 'hello']
```

With tuples, the assignment to x[2] would have failed.

We can also mutate multiple elements of a list at once, using the slice assignment. For instance:

```
>>> mynums = [2, 4, 8, 10]
>>> mynums2 = ['four', 'eight', 'ten']
```

```
>>> mynums[1:] = mynums2
>>> mynums
[2, 'four', 'eight', 'ten']
```

We can even use the `map` function in this manner:

```
>>> mynums = [2, 4, 8, 11]
>>> mynums[1:] = map(lambda x: 2*x, mynums[1:])
>>> mynums
[2, 8, 16, 22]
```

As a neat little trick, we can check quickly in an if statement whether a tuple is empty:

```
>>> x = tuple()
>>> if x:
...     print('I'm empty!')
>>> if not x:
...         print('But I still exist!')
But I still exist!
```

We can check if a list is empty in the same way.

1.1 Questions

For the following questions, you will define a function that deals with tuples, and then define a similar function that deals with mutable lists.

1. Define `append_tup(tup1, tup2)` that returns a tuple with the elements of tup2 appended to tup1.

   ```
   def append_tup(tup1, tup2):
   ```

   Solution:

   ```
       return tup1 + tup2
   ```

2. Now define `append_lst(lst1, lst2)` that returns lst2 appended to the end of lst1, using list mutation.

   For example,

   ```
   >>> x = [1,2,3,4]
   >>> y = [5,6,7,8]
   >>> append_mut(x, y)
   [1,2,3,4,5,6,7,8]
   ```

```
>>> x
[1,2,3,4,5,6,7,8]
```

```
def append_mut(lst1, lst2):
```

Solution:

```
    lst1[:] = lst1 + lst2
    return lst1
```

Try the rest of the questions recursively.

3. Define `filter_tup(pred, tup)` that returns a tuple with elements that do not satisfy pred filtered out.

```
def filter_tup(pred, tup):
```

Solution:

```
    if tup == ():
        return tup
    if pred(tup[0]):
        return (tup[0],) + filter_tup(pred, tup[1:])
    return filter_tup(pred, tup[1:])
```

4. Define `filter_mut(pred, lst)` that filters the elements out of lst, using list mutation.

```
def filter_mut(pred, lst):
```

Solution:

```
    if not lst:
        return lst
    elif not pred(lst[0]):
        lst[1:] = filter_lst(pred, lst[1:])
        lst.pop(0)
        return lst
    else:
        lst[1:] = filter_lst(pred, lst[1:])
        return lst
```

```
or, without using .pop:

def filter_mut(pred, lst):
    if not lst:
        return lst
    elif not pred(lst[0]):
        lst[:] = filter_lst(pred, lst[1:])
        return lst
    else:
        lst[1:] = filter_lst(pred, lst[1:])
        return lst
```

5. Define `map_tup(fn, tup)`.

```
def map_tup(fn, tup):
```

Solution:

```
    if tup == ():
        return ()
    return (fn(tup[0]),) + map_tup(fn, tup[1:])
```

6. Define `map_mut(fn, lst)`, using list mutation.

```
def map_mut(fn, lst):
```

Solution:

```
    if not lst:
        return lst
    else:
        lst[0] = fn(lst[0])
        lst[1:] = map_mut(fn, lst[1:])
        return lst
```

7. Define `interleave_tup(tup1, tup2)` which returns the elements of tup1 and tup2 interleaved.

```
def interleave_tup(tup1, tup2):
```

Solution:

```
if tup1 == ():
    return tup2
elif tup2 == ():
    return tup1
else:
    return tup1[0] + interleave_tup(tup2, tup1)
```

8. Define `interleave_mut(lst1, lst2)` which returns the elements of lst1 and lst2 interleaved, using list mutation. `lst1` should be mutated after the call. The value of `lst2` is not important.

```
def interleave_mut(lst1, lst2):
```

Solution:

```
if not lst1:
    lst1[:] = lst2
    return lst1
elif not lst2:
    return lst1
else:
    lst1[1:] = interleave_mut(lst2, lst1[1:])
    return lst1
```

2 List Comprehension

If we want a quick way to create a new list from a given sequence, we can use something called list comprehension. The syntax is as follows:

```
[<map expression> for <name> in <sequence>]
```

This will take an expression and a sequence, apply the function to every item in the sequence, and then create a list out of the result. Be careful! We're not passing in a function to the map expression, but rather something that evaluates to a value, like $x * 2$. We can also add a filter at the end:

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

This is saying perform the map expression on every element in the sequence that satisfies the filter expression. In other words, the filter expression comes before the map expression.

1. What would Python do?

```
>>> seq = range(10)
>>> x = [x * x for x in seq]
>>> x
```

> **Solution:**
>
> `[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]`

```
>>> [elem+1 for elem in x if elem % 2 == 0]
```

> **Solution:**
>
> `[1, 5, 17, 37, 65]`

3 Dictionaries

A mapping from keys to values is a very useful data structure in Computer Science. We have built IDicts out of tuples and defined selectors for our data type. Python has a built-in mapping, called a dictionary (or `dict`). Unlike the immutable IDict, the Python `dict` is mutable. We construct and use built-in Python dictionaries as follows:

```
>>> empty_dictionary = dict()
>>> empty_dictionary
{}
>>> dict = {'some_key': 1234, 'another_key': 8383, 28312: 1111}
>>> dict['some_key']
1234
>>> dict[28312]
1111
>>> dict['non_existant_key']
Traceback... KeyError: 'non_existant_key'
```

Note the use of curly braces, { }, to create the dictionary. We put key-value pairs in the form of key: value, and we use commas to separate each key value pair. To look up the value of a key, we use [ ] to index, similar to how we indexed into lists and tuples.

Also, notice how we were able to use both strings and numbers as keys. In fact, any *immutable* data type can be used as a key! This means that strings, numbers, tuples, and even functions could be used as keys into a dictionary. This also means that lists cannot be used as keys.

We can also modify the `dict` using the following syntax:

```
>>> dict['some_new_key'] = 5
>>> dict['some_new_key']
5
>>> dict['some_new_key'] = 123123
>>> dict['some_new_key']
123123
```

Notice how we were first able to define a new key-value mapping, and then also change the key-value mapping.

It turns out that the IDict methods we provided you are similar to ones that Python has built-in for its own dictionaries. For example, Python dictionaries have a `keys` method that will return a sequence of keys.

1. Write `make_inverse_dict(dict)` 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 = {'hope': 3, 'love': 2, 'pants': 3}
   >>> d2 = make_inverse_dict(d1)
   >>> d2 #note that we know nothing about the order of dictionaries
   {3: ('hope', 'pants'), 2: ('love',)}
   ```

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

   ```
   def make_inverse_dict(dict):
   ```

   Solution:

   ```
       inverse = {}
       for key in dict.keys():
           val = dict[key]
           if val in inverse:
               inverse[val] = inverse[val] + (key,)
           else:
               inverse[val] = (key,)
       return inverse
   ```

One more thing to keep in mind: dictionaries are $unordered$, meaning that you can never make any assumptions about the order in which keys or values are stored in Python dictionaries.