Mutability, Functions on Mutable Data, and Nonlocal 7

Computer Science 61A

July 10, 2014

1 Mutability

Let's say you order a mushroom and cheese pizza from Domino's. They represent your order as a list:

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

Five minutes later, you realize that you really want onions on the pizza. Based on all the rules we know so far, this means that Domino's would have to build an entirely new list to add onions:

```
pizza2 = pizza1 + ['onions']
```

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`.

It turns out 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')`. Now `pizza1` would be

```
['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.

1.1 What Would Python Output?

Consider the following definitions and assignments, and determine what Python would output for each of the calls below *if they were evaluated in order*.

1. ```
   >>> lst1 = [1, 2, 3]
   >>> lst2 = lst1
   >>> lst2 is lst1
   ```

2. ```
   >>> lst1.append(4)
   >>> lst1
   ```

3. ```
   >>> lst2
   ```

4. ```
   >>> lst2[1] = 42
   >>> lst2
   ```

5. ```
   >>> lst1
   ```

6. ```
   >>> lst1 = lst1 + [5]
   >>> lst1
   ```

7. ```
   >>> lst2
   ```

8. ```
   >>> lst2 is lst1
   ```

2 List Methods

List *methods* are functions tied to a specific list. They're called using *dot notation*, in the form `lst.method()`. Some common list methods:

```
lst.append(el) # Mutates lst to add el to the end

lst.insert(i, el) # Mutates lst to add el at index i

lst.sort() # Mutates lst to sort elements in place

lst.remove(el) # Mutates lst to remove the
# first occurrence of el in lst, otherwise errors

lst.index(el) # Returns the index of first occurence
# of el in lst, errors if el doesn't exist. DOES NOT MUTATE.
```

It is important to note that none of the mutating list methods actually *return* a new list - they simply modify the original list and return `None`.

2.1 List Mutation Questions

1. Write a function that removes all instances of `el` from `lst`.

```
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 `square_elements` which takes in a `lst` and replaces each element with the square of that element. *Make sure to mutate lst rather than returning a new list.*

```
def square_elements(lst):
    """
    >>> lst = [1, 2, 3]
    >>> square_elements(lst)
    >>> lst
    [1, 4, 9]
    """
```

3. Write a function which takes in a list `lst`, and two values `x` and `y`, and adds as many `y`s to the end of `lst` 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.
    >>> lst = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5]
    """
```

4. Write a function which reverses a list using mutation. Don't use the built-in method `reverse`.

```
def reverse_list(lst):
    """
    >>> lst = [1, 2, 3, 4]
    >>> reverse_list(lst)
    >>> lst
    [4, 3, 2, 1]
    >>> pi = [3, 1, 4, 1, 5]
    >>> reverse_list(pi)
    >>> pi
    [5, 1, 4, 1, 3]
    """
```

3 Higher-Order Functions in List Comprehensions

Often, we want to apply a function over all the elements of a list - for example, finding the sum or product of all the elements. One way to do this is by using the `reduce` function. To access it, use this `import` statement:

```
from functools import reduce
```

`reduce` is a higher-order function which takes in a function `accum`, a `lst`, and a `start` which is the same type of element as the elements in `lst`. Starting with the `start`, it repeatedly accumulates the elements of `lst` using the `accum` function. For example,

```
from operator import add
from functools import reduce
reduce(add, [i for i in range(5)], 100)
```

would return 110: starting with 100, it successively adds on 0, then 1, then 2, then 3, and finally 4.

Notice that we used a list comprehension above. Recall the syntax for list comprehensions:

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

Here the `if` clause is optional.

3.1 Reduce and List Comprehension Questions

1. Using list comprehensions, `reduce`, and `lambda` expressions, write the `factorial` function non-recursively in one line.

   ```
   factorial =
   ```

2. Using `reduce` and a `lambda` expression, write `max_even`, which takes in a list of positive numbers and returns the largest even number.

   ```
   max_even =
   ```

3. Write `money_left`, which takes in an `allowance` and a list `prices`, and returns the amount of money left if you start with `allowance` and successively subtract off each element in `prices`.

   ```
   money_left =
   ```

4. Challenging: Using list comprehensions, given `link` and an `is_prime` function, write a function which creates a `linked_list` of the squares of prime numbers from 2 to `n`. Hint: Be careful with order of operations - think about how subtraction worked in `money_left`

   ```
   primes_squared =
   ```

4 Dictionaries

Dictionaries are data structures which map keys to values. Dictionaries in Python are usually 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 key is in square brackets rather than the index.

4.1 What Would Python Print?

Assume these commands are entered in order after the above code has been executed in the interpreter.

1. `>>> 'mewtwo' in pokemon`

2. `>>> len(pokemon)`

3.
```
>>> pokemon['ditto'] = pokemon['jolteon']
>>> pokemon[('diglett', 'diglett', 'diglett')] = 51
>>> pokemon[25] = 'pikachu'
>>> pokemon
```

4.
```
>>> pokemon['mewtwo'] = pokemon['mew']*2
>>> pokemon
```

5.
```
pokemon[['firetype', 'flying']] = 146
```

Although dictionaries cannot use other dictionaries as keys, they can be arbitrarily deep, meaning the values of a dictionary can be themselves dictionaries. To traverse these deep dictionaries, we'll need to learn some more dictionary methods.

To iterate over a dictionary's keys, use

```
for key in dictionary.keys():
    # Stuff
```

To remove an entry in a dictionary, use

```
del dictionary[key]
```

To add `val` corresponding to `key` *or* to replace the current value of `key` with `val`, use

```
dictionary[key] = val
```

4.2 Dictionary Questions

1. 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(d, x, 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}}
    """
```

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}
    """
```

5 Nonlocal

The `nonlocal` keyword can be used to modify a variable in parent frame outside the current frame (as long as it's not the global frame). For example, consider `make_step`, which uses `nonlocal` to modify `num`:

```
def make_step(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step
```

5.1 Nonlocal Environment Diagrams

1. Draw the environment diagram for the following series of calls after `make_step` has been defined:

```
>>> s = make_step(3)
>>> s()
>>> s()
```

2. Given the definition of `make_buy_item` below,

```
def make_buy_item(total_gold):
    def buy_item(cost):
        nonlocal total_gold
        if total_gold < cost:
            return 'Go farm some more champions'
        total_gold = total_gold - cost
        return total_gold
    return buy_item
```

draw an environment diagram for the definition as well as the following series of commands:

```
>>> bloodthirster, zeal, total_gold = 3500, 1100, 3800
>>> shopkeeper = make_buy_item(total_gold)
>>> shopkeeper(bloodthirster)
>>> shopkeeper(zeal)
```

5.2 Nonlocal Misconceptions

For each of the following pieces of code, explain what's wrong with the use of nonlocal.

1. ```
   a = 5
   def add_one(x):
       nonlocal x
       x += 1

   >>> add_one(a)
   ```

2. ```
   def another_add_one():
       nonlocal a
       a += 1

   >>> another_add_one(a)
   ```