NONLOCAL AND MUTABILITY 7

COMPUTER SCIENCE 61A

July 14, 2015

1 Mutating Lists

Many of Python's primitive types are considered *immutable*, meaning that once they have been created, their value *cannot* change. Examples of these immutable types include strings, tuples, and numbers.

However, lists, dictionaries, and some other data types that are considererd *mutable*, meaning the values of a specific instance or object of that type *may change*.

Imagine you go to *CREAM* on Telegraph Avenue and you order an ice-cream sandwich. Suppose *CREAM* chooses to represent your order as a list like so:

```
>>> sandwich = ['ice-cream', 'cookie']
```

Suppose that, while *CREAM* was preparing your order, you decide you want to top your sandwich with sprinkles. Without mutation, CREAM changes your order like so:

```
# creates a new python list
>>> new_sandwich = sandwich + ['sprinkles']
>>> new_sandwich
['ice-cream', 'cookie', 'sprinkles']
>>> sandwich # the original list is unmodified
['ice-cream', 'cookie']
```

What was the point of *CREAM* having to make an entirely new sandwich just to add sprinkles? They could have simply modified the original sandwich! That's what mutation is all about! Instead, they could have done:

```
>>> sandwich.append('sprinkles') # mutates original list
>>> sandwich
['ice-cream', 'cookie', 'sprinkles']
```

1.1 What Would Python Print?

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 List Methods

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`.

2.1 Code Writing Questions

1. Write a function `square_elements` which takes a `lst` and replaces each element with the square of that element. *Mutate* `lst` *rather than returning a new list*.

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

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. 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]
    """
```

4. 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]
    """
```

3 Nonlocal

Until now, you've been able to access variables in parent frames, but you have not been able to modify them. The `nonlocal` keyword can be used to modify a variable in the parent frame outside the current frame (as long as it's not the global frame). For example, consider `stepper`, which uses `nonlocal` to modify `num`:

```
def stepper(num):
    def step():
        nonlocal num  # declares num as a nonlocal variable
        num = num + 1 # modifies num in the stepper frame
        return num
    return step
```

3.1 Environment Diagrams

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

```
s = stepper(3)
s()
s()
```

2. Given the definition of `make_shopkeeper` below, draw the environment diagram.

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

infinity_edge, zeal, gold = 3800, 1100, 3800
shopkeeper = make_shopkeeper(gold - 1000)
shopkeeper(zeal)
shopkeeper(infinity_edge)
```

3.2 Some Common Misconceptions

1. What is wrong with the following code?

```
a = 5
def adder(x):
    def add(y):
        nonlocal x, y
        x += y
        return x
    return add
adder(2)(3)
```

2. What is wrong with the following code?

```
a = 5
def another_add_one():
    nonlocal a
    a += 1
another_add_one()
```