1 Aliasing and mutable objects

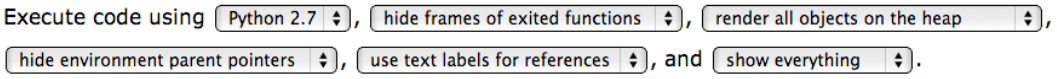
Aliasing is when two or more variables refer to the same object. This is not a new concept for us... however, things get more interesting when the object is *mutable*.

For example, `L2` is an alias of `L` (and vice versa) because they both refer to the same list object.

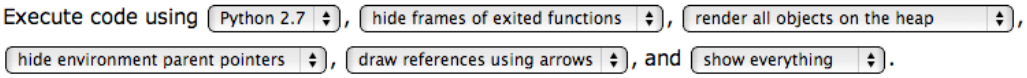
```
>>> L = [1,2,3]
>>> L2 = L
>>> print L2
[1, 2, 3]
>>> L[1] = 200
>>> print L
[1, 200, 3]
>>> print L2      # since L2 is an alias, the change to L affects L2!
[1, 200, 3]
```

Notice that even though we did not do anything with the variable `L2`, it is clearly affected by operations performed on its alias `L`. Since lists are *mutable*, when you change `L`, you are also changing `L2` since they both refer to the same object. Strings can be aliased too but since they are not mutable, we tend not to worry about the fact that two variables might refer to the same string object.

The python visualizer is a great tool for understanding aliasing (`http://www.pythontutor.com/visualize.html`). Be sure to adjust the settings so they look like this:



or this



There's more on the back!

2 Aliasing and functions

Aliasing arises with functions. The *parameter* of a function is always an alias of the variable that is passed in as an *argument*. In this example, the parameter `L` is an alias of the argument `a_list`.

```
def double(L):
    '''
    (list of int) -> NoneType
    Doubles every number in list L
    '''
    for i in range(len(L)):
        L[i] = L[i] * 2
    print L

a_list = [5, 6]
double(a_list)
print a_list
```

If we run the above code, the list `[10, 12]` is printed twice (once for **`print`** `L` and once for **`print`** `a_list`). That's because...

- `L` and `a_list` are aliases: they refer to the same list object
- the function mutates the list object: each value in the list is doubled

3 Avoiding aliasing

To avoid aliasing, you can make a copy of the list (using slicing). In this example, `L` and `L2` are not aliases.

```
>>> L = [1,2,3]
>>> L2 = L[:]       # makes a copy
>>> L[1] = 200
>>> print L
[1, 200, 3]
>>> print L2
[1, 2, 3]
```

Subtle detail: if you are an eagle-eyed observer, you might notice that while these two lists are not aliases, the contain the same collection of items. If those items are mutable (e.g., the item is itself a list) then mutating one of the items in `L` would cause a change to that item in `L2`. If you do not understand this subtle detail, do not worry about it at this point.

Lecture #17 handout introduced lists. This handout describes list methods.

1 Lists vs. strings

We have compared lists and strings before. Here are some new similarities and differences. List also support slicing and it works exactly the same way as it does on strings (Handout #19). Lists and strings have some methods in common, such as `count`. Some methods are different: for instance `find` on a string works differently than `index` on a list. Another significant difference: lists have methods that *mutate* the list object. The table below indicates which methods actually change or mutate the list. Finally, you can translate from lists to strings and back. To turn a string into a list, use the **`list`** function. To turn a list of characters into a string, use the `join` method.

```
>>> s = "hello"
>>> L = list(s)
>>> L
['h', 'e', 'l', 'l', 'o']
>>> L[0] = 'c'
>>> s2 = ''.join(L)
>>> s2
'cello'
```

2 List methods

List Method	Arguments	Description	Mutates?
`append`	`x`	Adds item `x` to the *end* of the list.	Yes!
`extend`	`other_list`	Adds all of the items in list `other_list` to the end of the list.	Yes!
`remove`	`x`	removes `x` from the list and returns nothing.If `x` is not in the list, you get an **error**.	Yes!
`pop`	`i`	Returns the item at index `i` and also removes it from the list.	Yes!
`count`	`x`	Returns the number of times item `x` occurs in the list	
`index`	`x`, `start`, `end`	Returns index of item. If `x` not in list, you get an an **error**! Note difference from `find` method on strings. The start and end parameters are *optional*.	
`insert`	`i`, `x`	Moves items at indexes `i` and larger to the right and inserts `x` in list at position `i`.	Yes!
`reverse`	none	Reverses the list. This method returns `None`.	Yes!
`sort`	none	Sorts the list. This method returns `None`.	Yes!

```
>>> L = ['a', 'b', 'c', 'b', 'd', 'b']
>>> L.index('c')
2
>>> L.index('z')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'z' is not in list
>>> L.count('z')
```

```
0
>>> L.count('b')
3
>>> L.pop(2)
'c'
>>> L
['a', 'b', 'b', 'd', 'b']
>>> L.remove('d')
>>> L
['a', 'b', 'b', 'b']
>>> L.remove('z')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>> L.append('A')
>>> L.insert(1, 'z')
>>> L
['a', 'z', 'b', 'b', 'b', 'A']
>>> L.extend(['b', 'c'])
>>> L
['a', 'z', 'b', 'b', 'b', 'A', 'b', 'c']
>>> L.sort()  # sort mutates list and returns None
>>> L
['A', 'a', 'b', 'b', 'b', 'b', 'c', 'z']
>>> L = L.sort()    # WRONG!  sort does NOT return the sorted list
>>> print L         # L is now None!
None
```

3 Exercises

Some solutions are presented in class and also included in the moodle version of this handout.

1. Write a function `remove_all` that takes that takes `L`, a list of ints, and an int `x` and removes all occurrences of `x` from the list. Hint: use `count`, `remove`, and a while loop. **Important point:** if you want to repeatedly *add/remove* items from a list, it's *not* a good idea to use a **`for`** loop over that list. (Essentially a for loop assumes the thing it's looping over is *not* changing.)

2. Write a function `remove_less_than` that takes `L`, a list of ints, and an int `x` and removes any occurrence that is strictly less than `x`. Again, use a while loop.

3. Write a function `is_anagram` that takes two strings and returns `True` if the strings are anagrams of each other. Challenge edition: ignore spaces and be case-insenstive. So `'Dormitory'` is an anagram of `'Dirty Room'`.

4. Write a function `count_distinct` that takes a list and returns the number of distinct items. For example, on `[10, 20, 10, 30, 20]` it should return 3 because there are three distinct items.