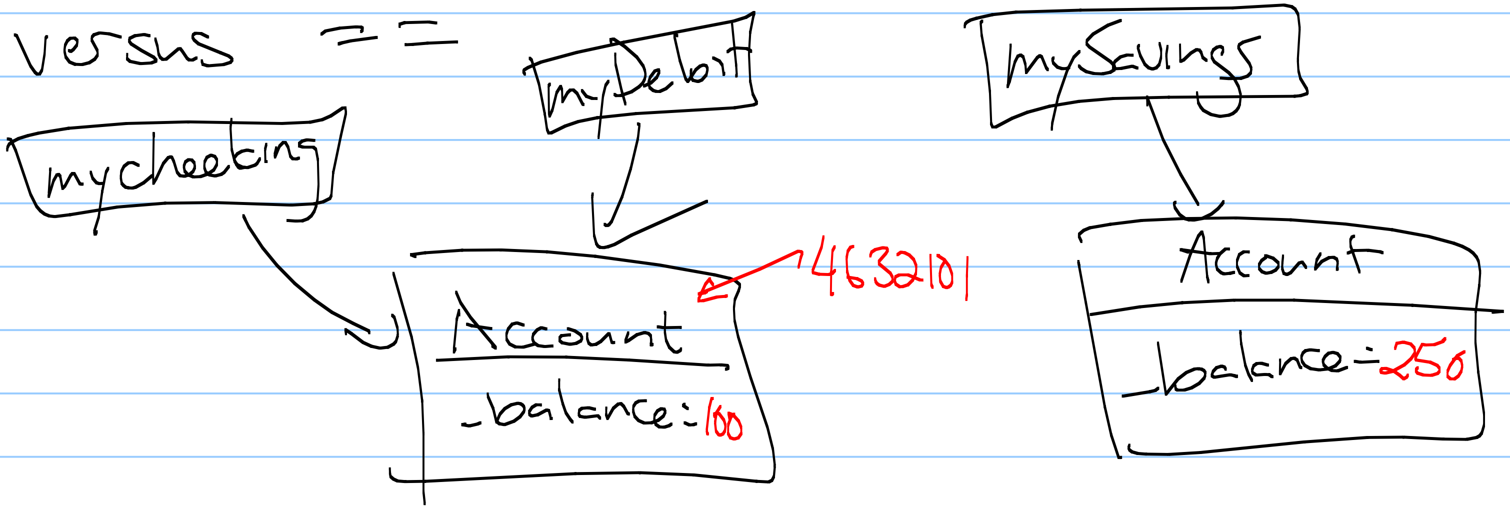
CS150 - More on Objects

Note Title 3/27/2012

- HW due Friday

Last time

- Basic Account class (for our example)
- id(variable) : returns a # which is a unique identifier for underlying object
- is versus ==



Primitive Types

Lists: Say we create 2 lists, which have the same contents.

Will they be the same object?

No – different ids

Strings: Will 2 identical strings have the same id? → Yes

(Test!)

`raw_input() == 'yes'`

Mutable versus immutable

The difference is in the type of operations.

If one list is changed, the other should not change.

But – strings are immutable! We'll never be able to change that object.

Some variation here:

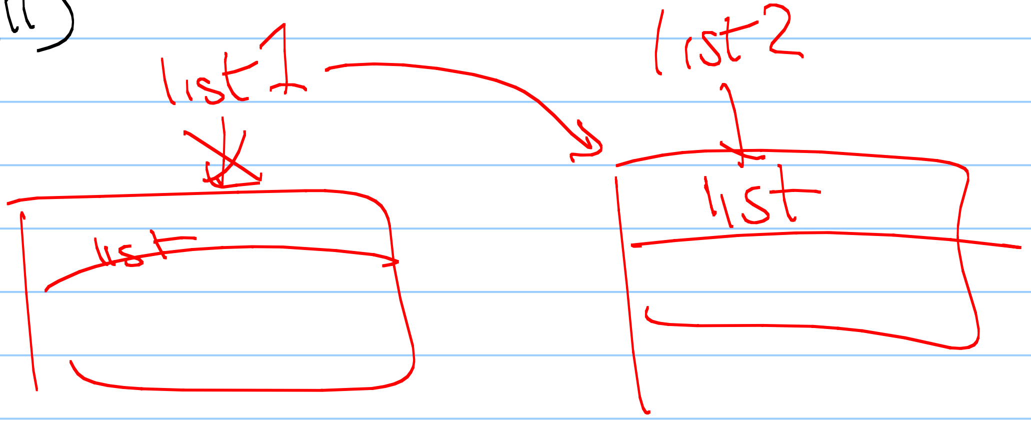
```
id(4)
id(2+2)
id('aloha') == id('Aloha'.lower())
```

Garbage collection

Creating an object allocates space in memory.

What happens to that data once we are done with it?

Ex: list1 = range(10)
list2 = range(11)
list1 = list2



Garbage Collection

The task of deallocating memory that is no longer used is called garbage collection.

Python does this for you:

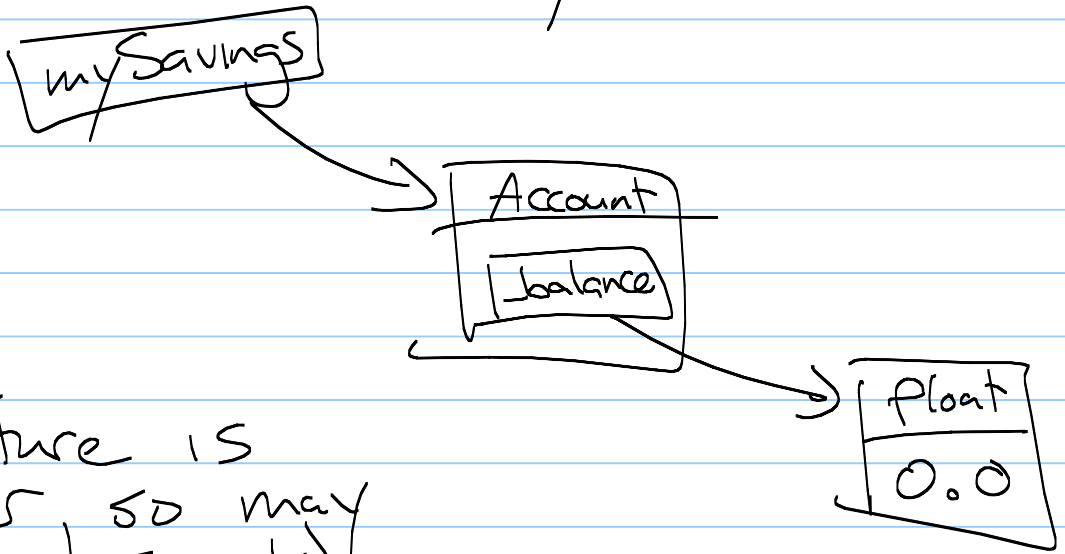
- each object keeps a reference count
- when ref count = 0, deletes that object

(This takes time, & is one of the reasons Python is a slower language than some others.)

Objects referencing other objects

Technically, most of our classes <u>reference</u> other objects.

So our Account really looks like:



(other picture is simpler, so may still use it)

Caution:

Immutable objects can contain mutable ones.

Ex: frozenAssets = (mySavings, myChecking)

Can we change the Accounts?

tuple is immutable

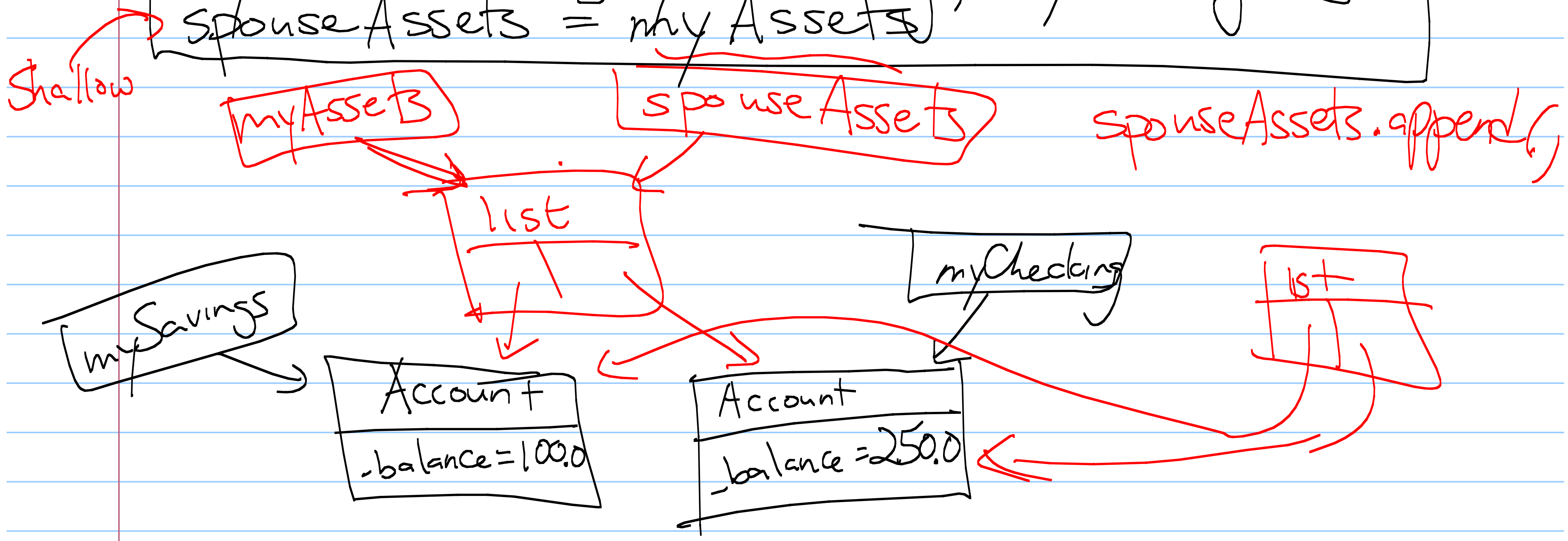
but Accounts are mutable!

```
mySavings.deposit(100)
frozenAssets[1].deposit(50)
```
} both ok

More cautions with aliasing: Shallow copy

Say we have:

```
myAssets = [myChecking, mySavings]
spouseAssets = myAssets
```



Another case:

If we have a command like:

```
spouseAssests.append(spouseRetirement)
```

This will change our list, too!

To keep our list unchange, need to actually create a second list.

loop to append things to a new list

Shallow versus deep copies

In a shallow copy, the attributes of the object reference the same objects as the original.

In a deep copy, the attributes are independent copies of the original.

Previous examples were both shallow.

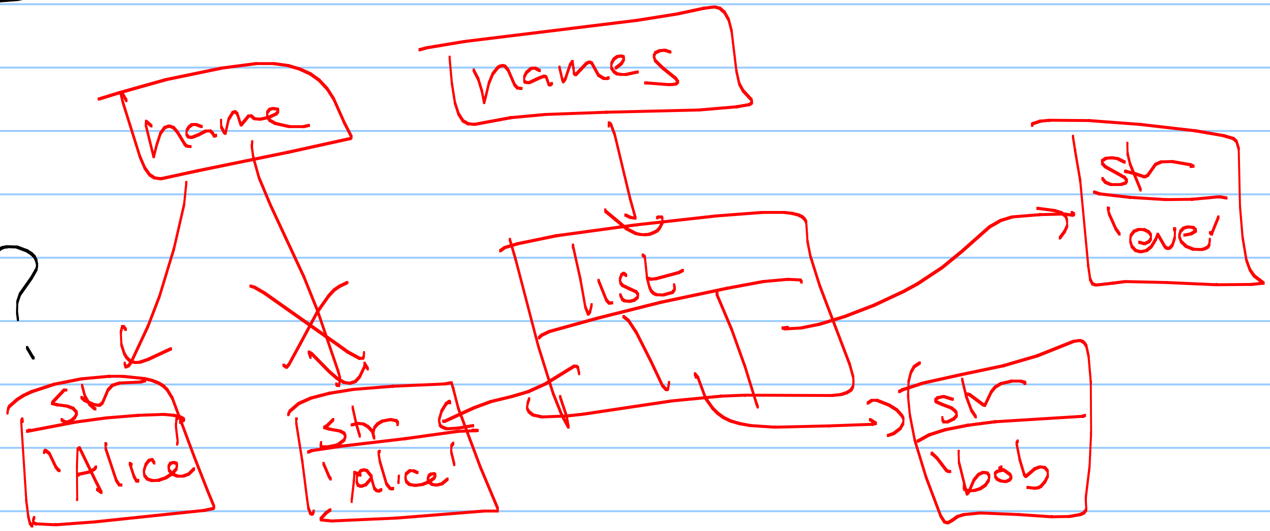
Example:

```
names = ['alice', 'bob', 'eve']
for name in names:
    name = name.capitalize()
print names
```

```
for i in range(len(names)):
    names[i] =
      names[i].cap'to..
```

Output?

How to fix?



Now:

Suppose we want original list unchanged:

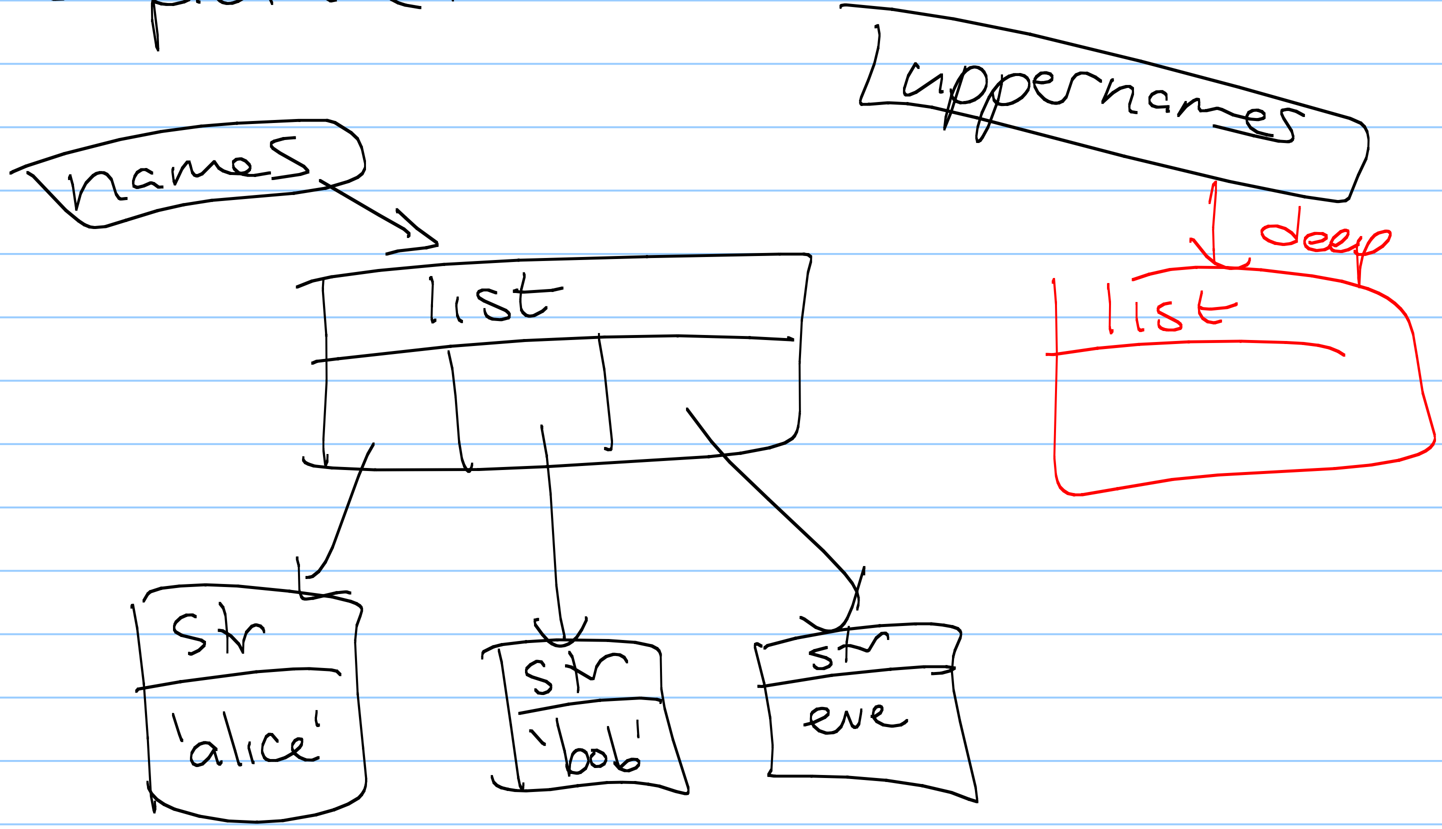
```
names = ['alice', 'bob', 'eve']
uppernames = list(names)   <- deep copy
for i in range(len(uppername)):
    uppernames[i] = uppernames[i].capitalize()

print names
```

Output? (+ why?)

```
['alice', 'bob', 'eve']
```

The picture:



To fix:

Make a deep copy:

```
uppernames = []
for name in names:
    uppernames.append(name.capitalize())
```

Safe way to make a deep copy.

Copy + deepcopy

Python has 2 modules,
copy + deepcopy
copy (x) → shallow copy

Caution: not allowed on some objects (like files)

But only gives deep (or shallow) 1 level down.

Next: Functions

```
def multiply(value, inputlist):
    ~~for item in inputlist:~~
        ~~item *= value~~
    for i in range(len(inputlist)):
        inputlist[i] *= value
```

Q: is inputlist changed outside the function?

Practice 10.1 & 10.2