Namespaces and Scope

Namespaces

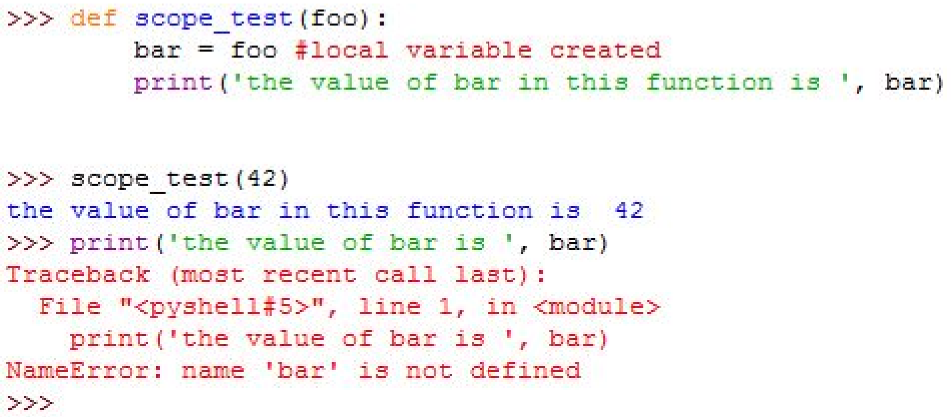
- You can think of a namespace as where a name is valid and can be used
    - If one function has a variable, another function (usually) cannot gain access to its value
    - Two functions can use variables with the same name. These are two separate variables.
    - Variables declared within functions go away after the function ends

Variable name resolution

- Namespace similar to a dictionary
- Multiple namespaces exist
- If a variable is not found in the local namespace, Python applies a name resolution algorithm, checking a sequence of namespaces:
    - **L**ocal
    - **E**nclosed
    - **G**lobal
    - **B**uilt-in

Local scope

- The set of program statements over which a variable exists, that it, can be referred to
- Local scope: a variable can be referenced only within the suite of the function where it was assigned



```
>>> def scope_test(foo):
        bar = foo #local variable created
        print('the value of bar in this function is ', bar)


>>> scope_test(42)
the value of bar in this function is  42
>>> print('the value of bar is ', bar)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    print('the value of bar is ', bar)
NameError: name 'bar' is not defined
>>>
```

Global variables

- A global variable is declared outside of any function and can be seen by any function…
  - …but cannot be changed by any function unless you use the “global” keyword
- Why don’t we want to use global variables very much?

How Python stores information

- *Objects* are Python's abstraction for data.
- All data in a Python program is represented by objects or by relations between objects.
- Every object has:
  - an identity (**Where** it is in memory. Unchangeable)
  - a type (**How to interpret** memory. Unchangeable)
  - a value (**What** is in memory. May (not) be changeable)

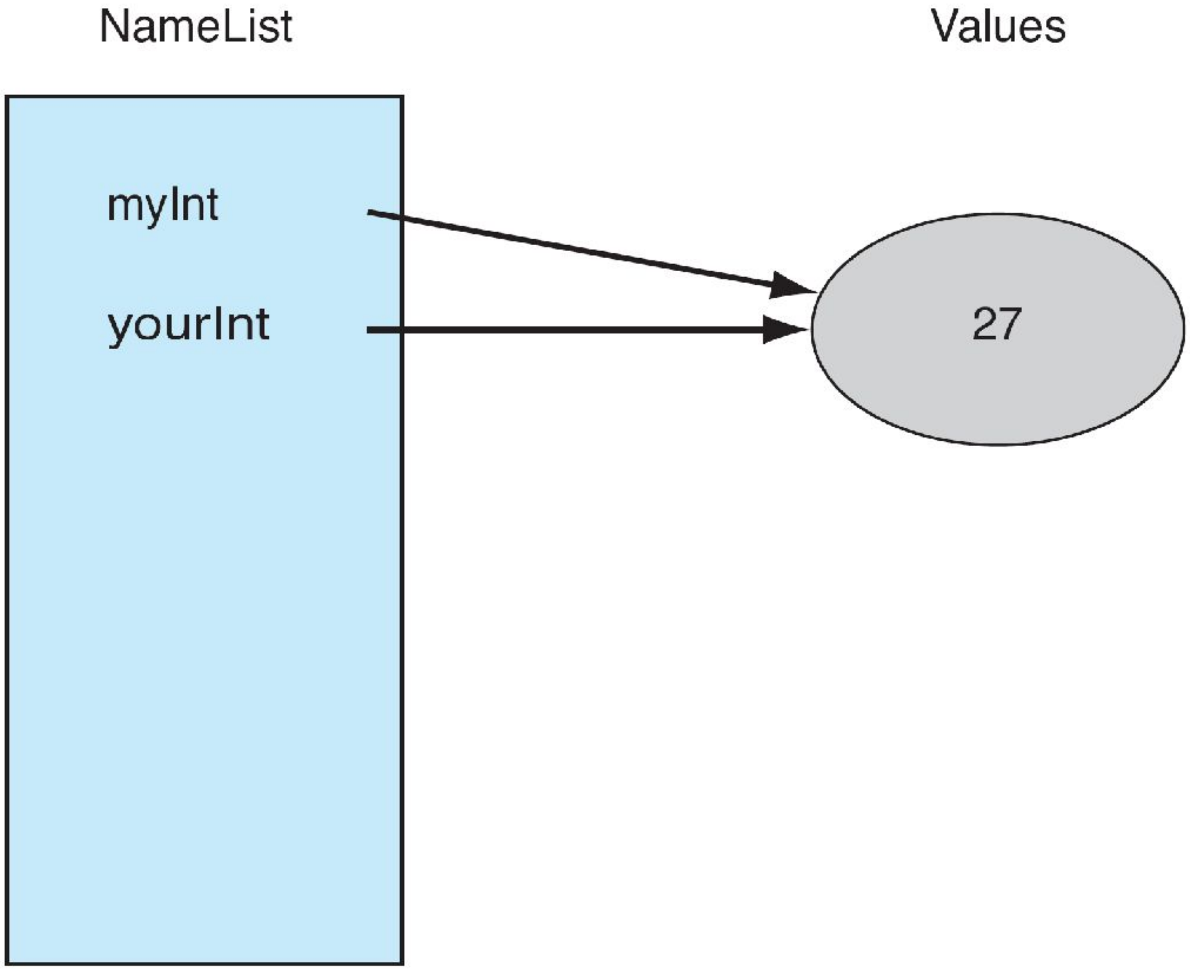
Parameter passing

- Let’s take a look at what happens when we try to pass mutable or immutable variables into functions…

Reminder: Assignment

- Assignment takes an object (the final object after all operations) from the right-hand-side and associates it with a variable on the left-hand side.
- When you assign one variable to another, you **share the association** with **the same object**.

```
myInt = 27
yourInt = myInt
```



NameList

Values

myInt

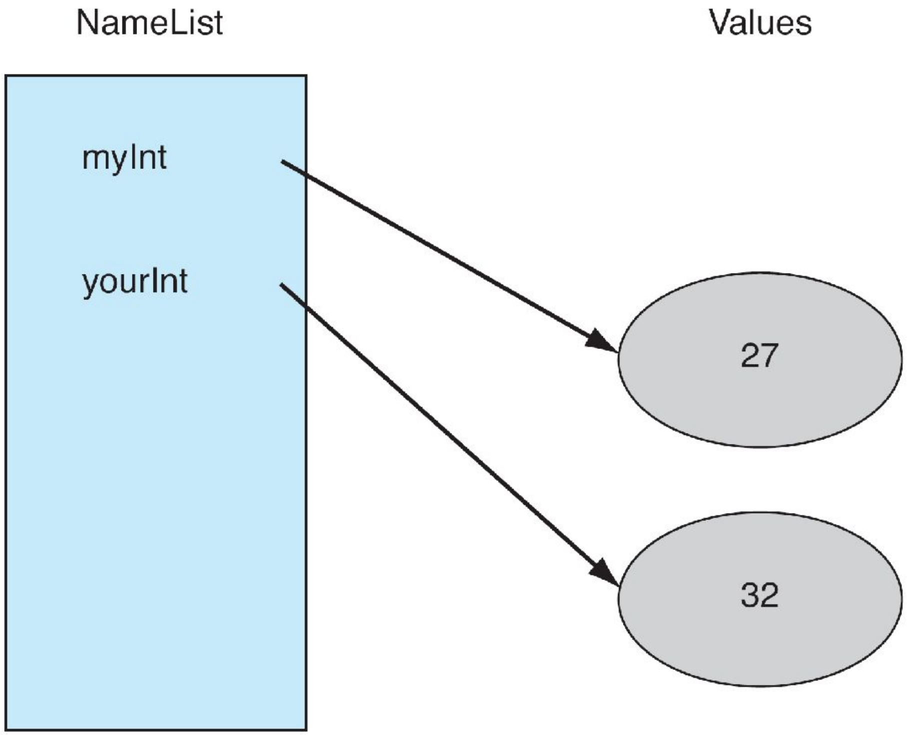
yourInt

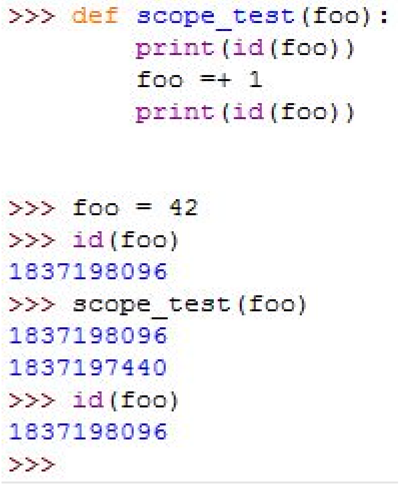
27

Immutables

- Object sharing, two variables associated with the same object, is not a problem since the object cannot be changed.
- Any changes that occur generate a **new** object.

```
myInt = 27
yourInt = myInt
yourInt = yourInt + 5
```





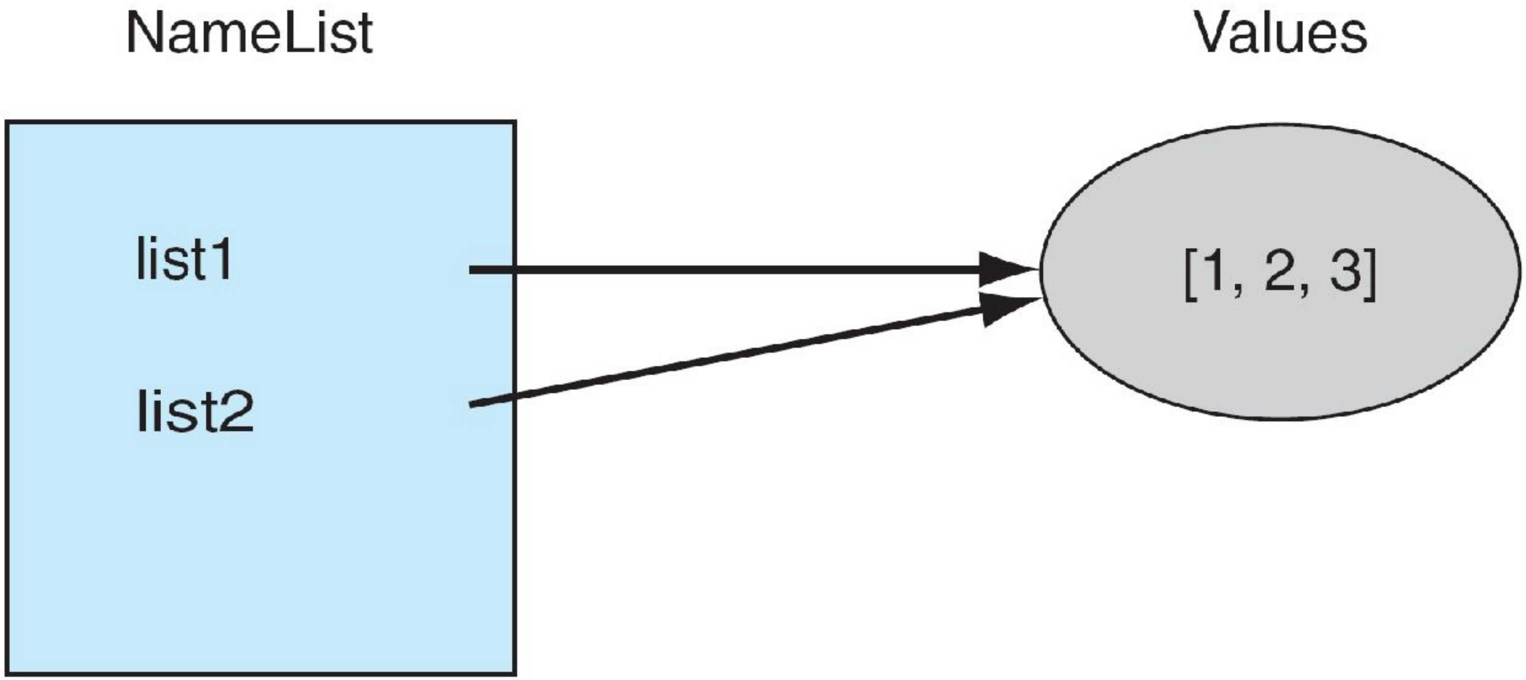
```
>>> def scope_test(foo):
        print(id(foo))
        foo =+ 1
        print(id(foo))


>>> foo = 42
>>> id(foo)
1837198096
>>> scope_test(foo)
1837198096
1837197440
>>> id(foo)
1837198096
>>>
```

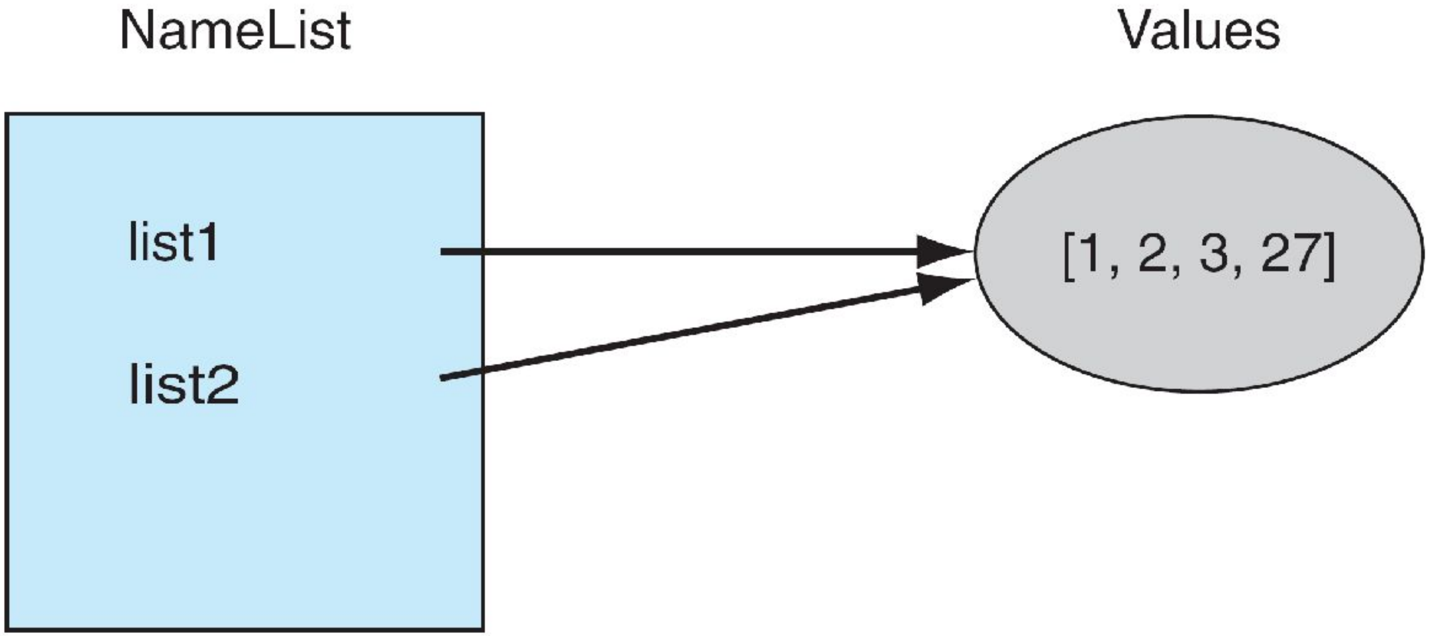
Mutability Changes an Object

- If two variables associate with the same object, they **both reflect** any change to that object.

```
list1 = [1,2,3]
list2 = list1
```



```
list1 = [1,2,3]
list2 = list1
list1.append(27)
```

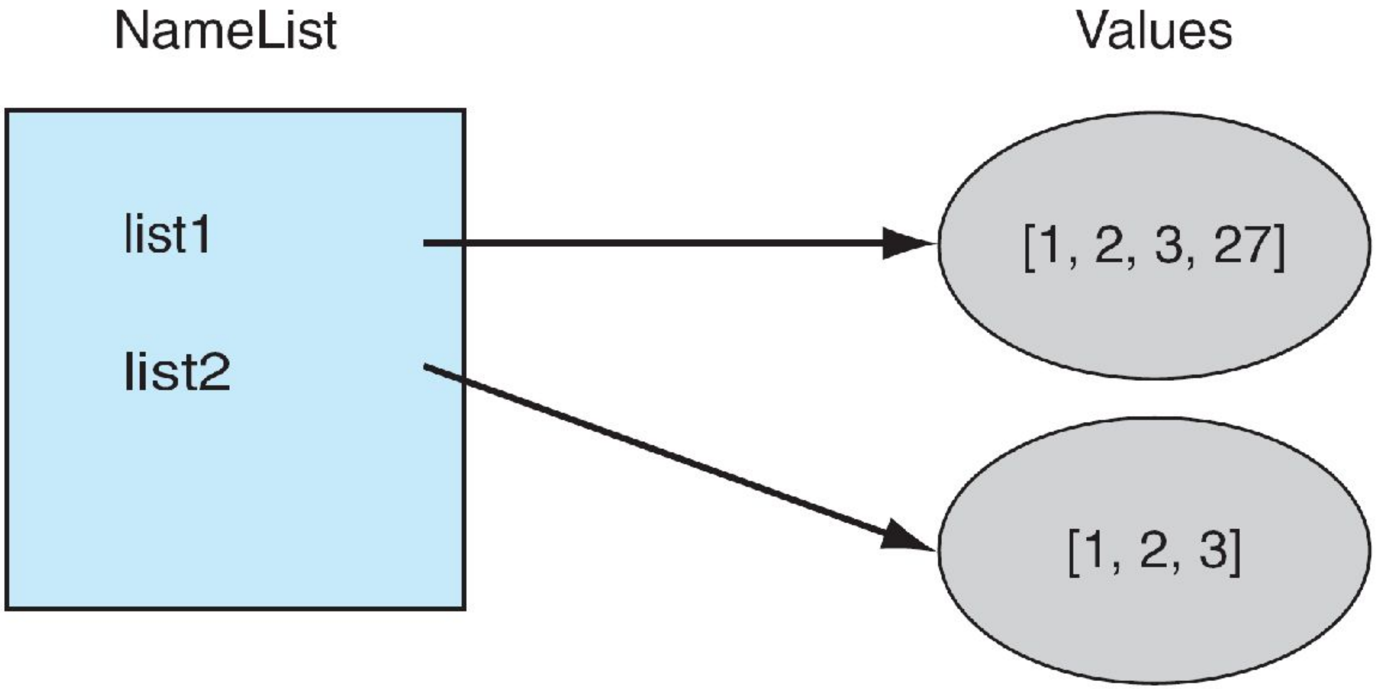


Copying

If we copy, does that solve the problem?

```
myLst = [1, 2, 3]
newLst = myLst[:]
```

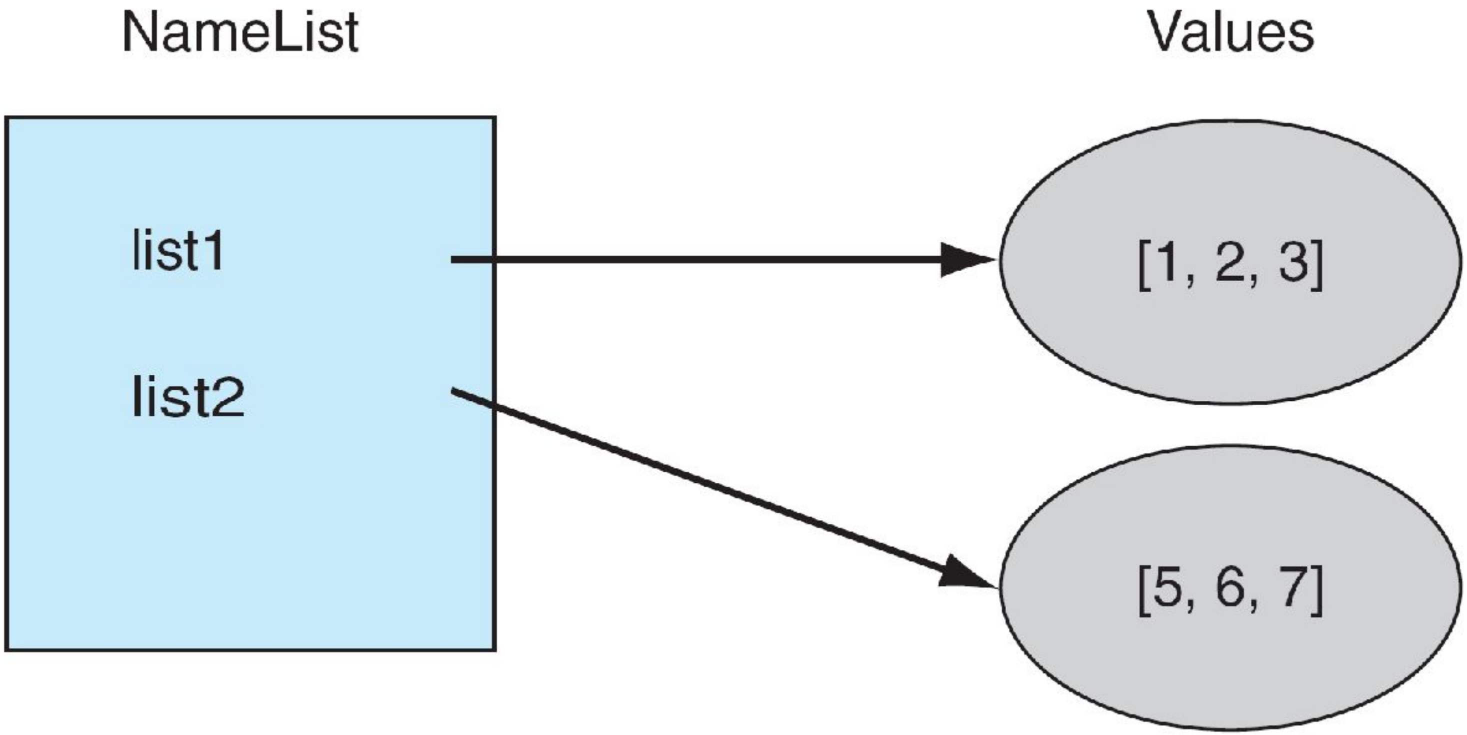
```
list1 = [1,2,3]
list2 = list1[:] #explicitly make a distinct copy
list1.append(27)
```



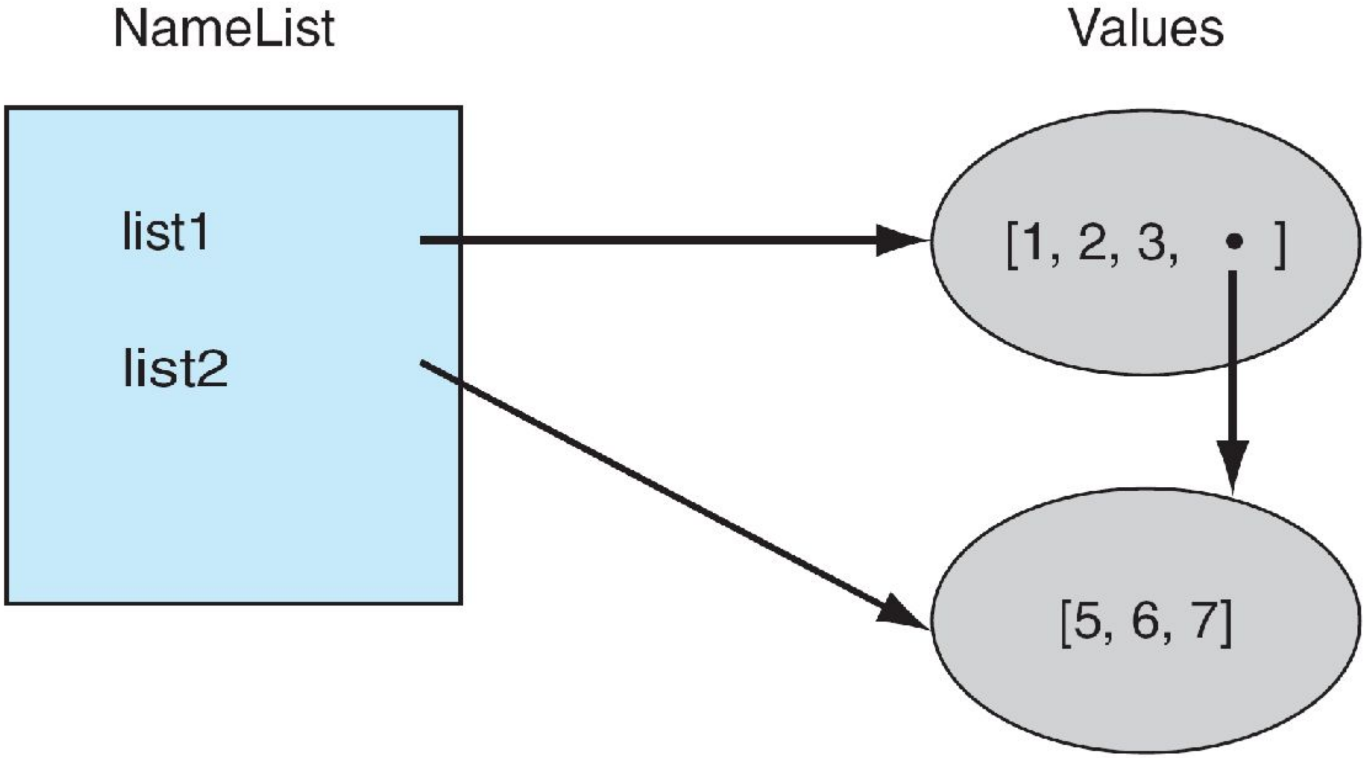
The Problem is What Gets Copied…

- The elements of the list are copied, but sometimes the elements of the list themselves are ***references*** (or associations).
- If the list has nested lists or uses other associations, the association gets copied. This is termed a **shallow copy**.

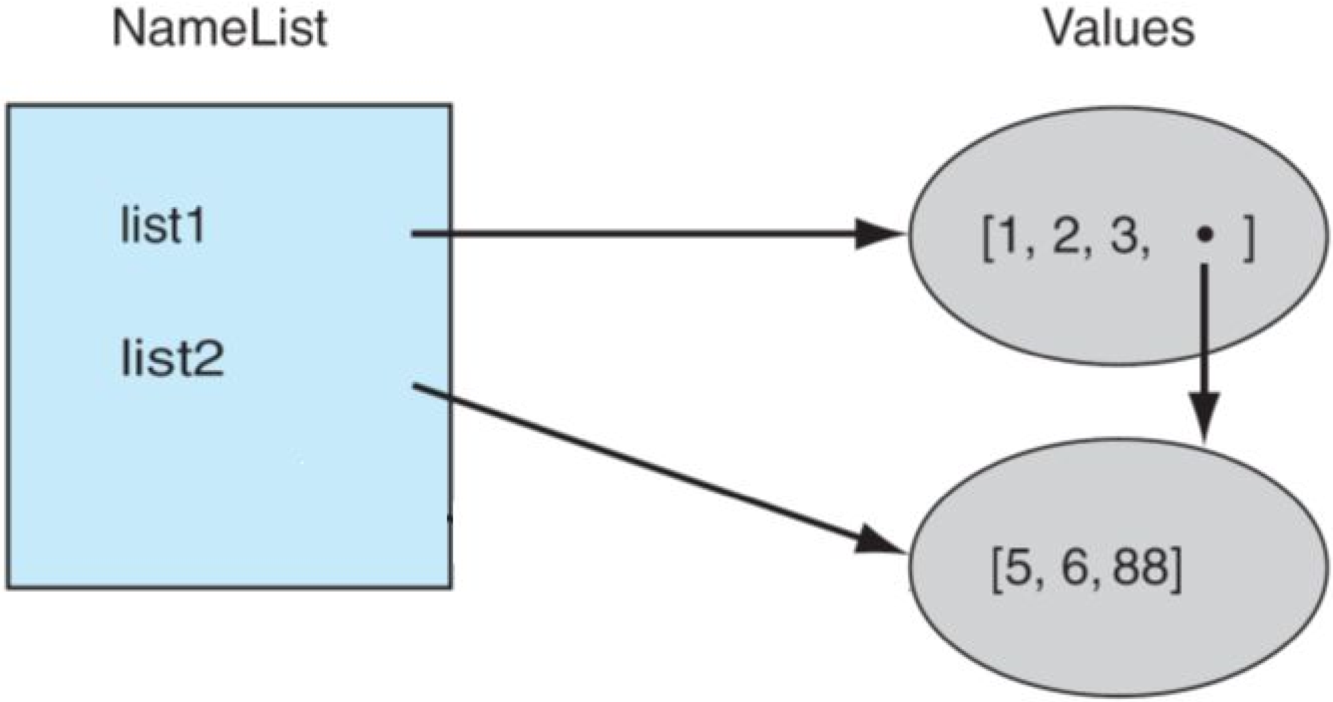
```
list1 = [1,2,3]
list2 = [5,6,7]
```



```
list1 = [1,2,3]
list2 = [5,6,7]
list1.append(list2)
```



```
list1 = [1,2,3]
list2 = [5,6,7]
list1.append(list2)
list2[2] = 88
```



Concluding notes…

- Managing complexity
- Principle of information hiding (encapsulation)
- All variables must be local (in 99.99% of cases)