Lecture 3.3 : Shallow and deep copies

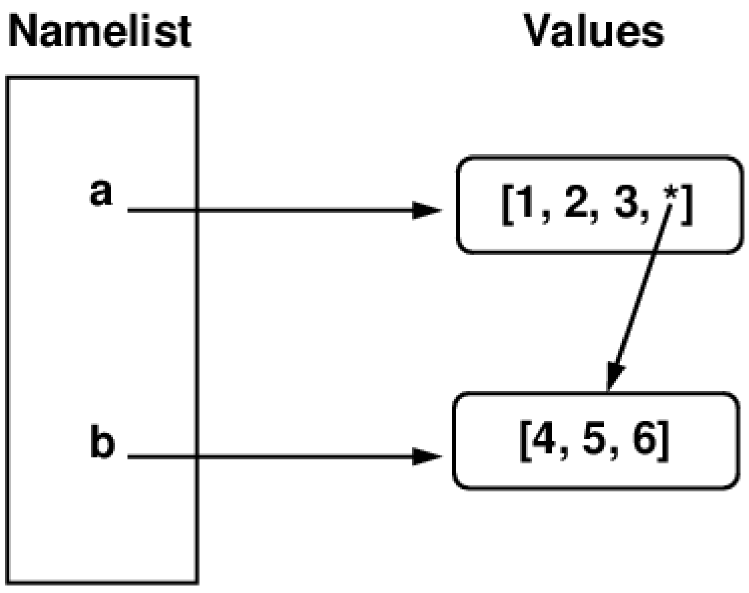
Introduction

- We conclude our exploration of the relation between variables, references and immutable/mutable objects with another example.

```
a = [1, 2, 3]
b = [4, 5, 6]
a.append(b)
print(a)
```

```
[1, 2, 3, [4, 5, 6]]
```

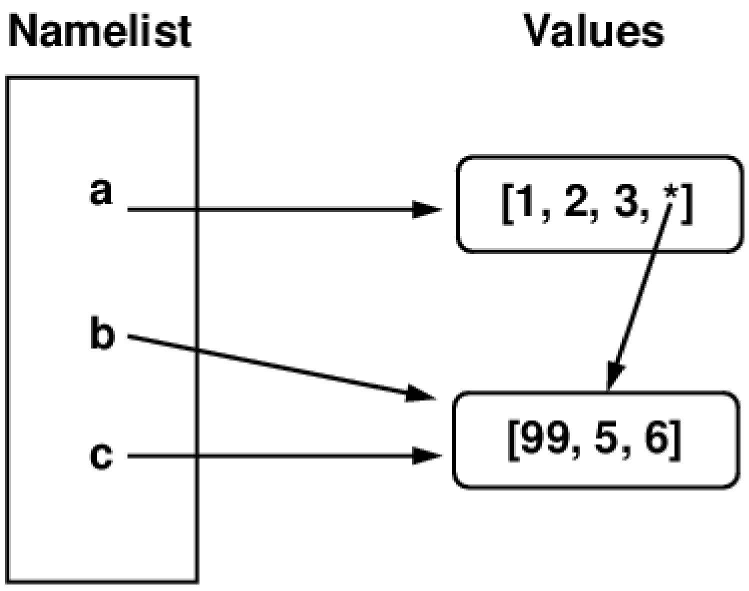
- What is going on here? The diagram below depicts the situation.



- As we can see, when we `a.append(b)` we append a *reference* to `b` to `a`. (To join list `b` to list `a` we would write `a.extend(b)`.)
- Thus after the append operation, `a` contains three integers (really it contains three references to immutable integers) and a reference to the mutable list referenced by `b`.
- The list `[4, 5, 6]` is thus shared by `a` and `b`. Any change to `b` affects `a` as the following example demonstrates.

```
c = b
c[0] = 99
print(c)
print(b)
print(a)
```

```
[99, 5, 6]
[99, 5, 6]
[1, 2, 3, [99, 5, 6]]
```



- It is crucial to note that `a.append(b)` adds a reference to `b` to `a`. It does *not* append a reference to *new copy* of the object referenced by `b` to `a`.

Shallow copies

- Suppose we wish to copy the list `a` above such that we create new copies of the objects it references rather than duplicating them. Let's try to do it with the slice operator:

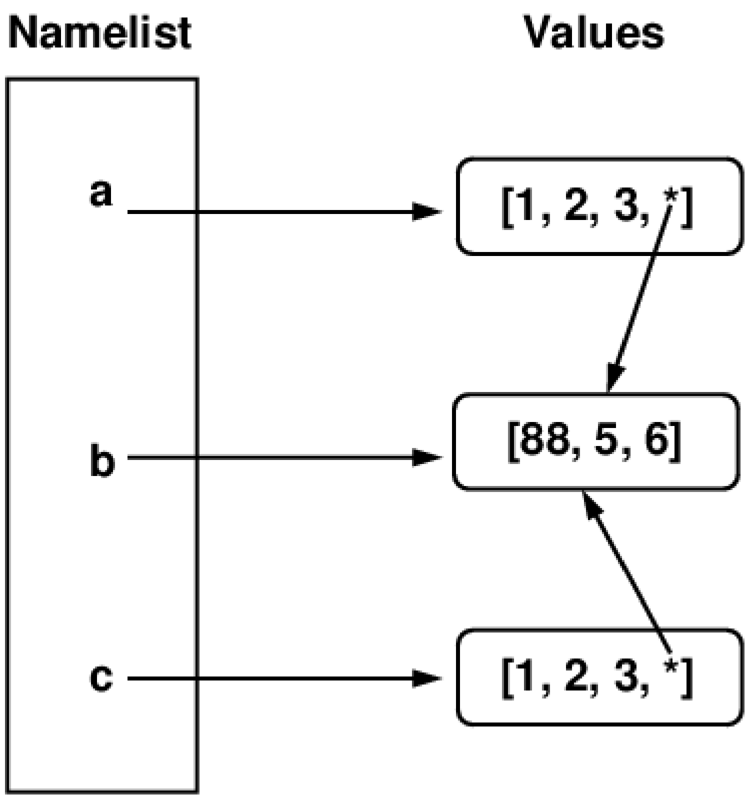
```
a = [1, 2, 3]
b = [4, 5, 6]
a.append(b)
c = a[:] # make c a reference to a copy of a
print(a)
print(c)
```

```
[1, 2, 3, [4, 5, 6]]
[1, 2, 3, [4, 5, 6]]
```

```
b[0] = 88
print(a)
print(c)
```

```
[1, 2, 3, [88, 5, 6]]
[1, 2, 3, [88, 5, 6]]
```

- That didn't work. What's going on? When we wrote `c = a[:]` the reference to `b` in `a` was copied to `c`. Thus the subsequent change to `b` affected both `a` and `c`.



- When we copy an object where we copy only references it contains and not the referenced objects themselves we are making a *shallow copy*. When we write `c = a[:]` we are making a shallow copy.

Deep copies

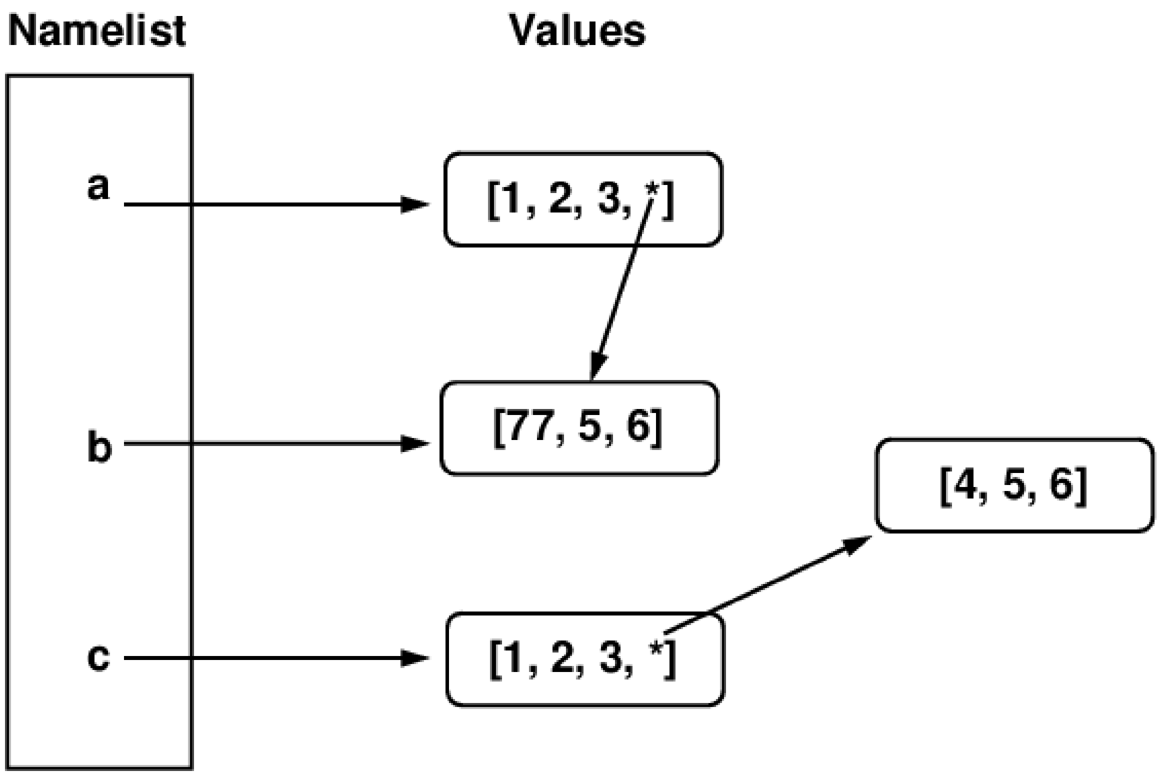
- Suppose we want to make a copy not of the references but the actual objects referenced? How can we do that? How can we make such a *deep copy*?
- Well, in the `copy` module there is a `deepcopy()` function that allows us to do just that. Below we illustrate it in action.

```python
a = [1, 2, 3]
b = [4, 5, 6]
a.append(b)
print(a)
print(b)
```

```
[1, 2, 3, [4, 5, 6]]
[4, 5, 6]
```

```python
from copy import deepcopy
c = deepcopy(a)
b[0] = 77
print(a)
print(c)
```

```
[1, 2, 3, [77, 5, 6]]
[1, 2, 3, [4, 5, 6]]
```



- Above we see that the list referenced by `c` does *not* contain a reference to `b` (it is unaffected by `b[0] = 77`). Instead `c` contains a reference to a new and separate copy of the list referenced from `a`.
- Note this form of copying is slower than the usual approach since it involves following all references in an object and creating new copies of the referenced objects.
- However, where independence from the source object is required in the copied object, it is a deep copy that must be implemented.