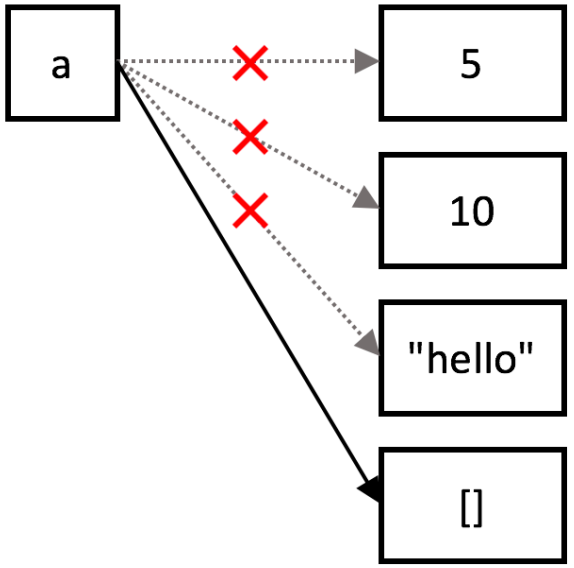
References, Aliases, and Mutable and Immutable Types

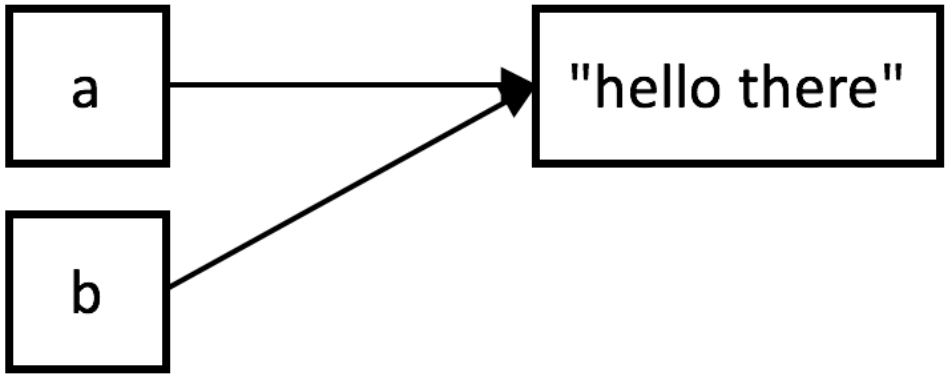
References

```
## each time 'a' is
assigned, it just points
## to a new value
a = 5
a = 10
a = 'hello'
a = []
```



- Variables are ‘names’ that refer to a value (object).
- Assigning just changes the reference.
- Objects with no in-arrows can be garbage collected.

Aliases




```
## when one name is assigned to another, the
## reference is copied. not the value
a = 'hello there!'
b = a
```

- When variables are assigned, the reference is copied, not the value (object).
- `a` and `b` are now aliases to the same `str` object.

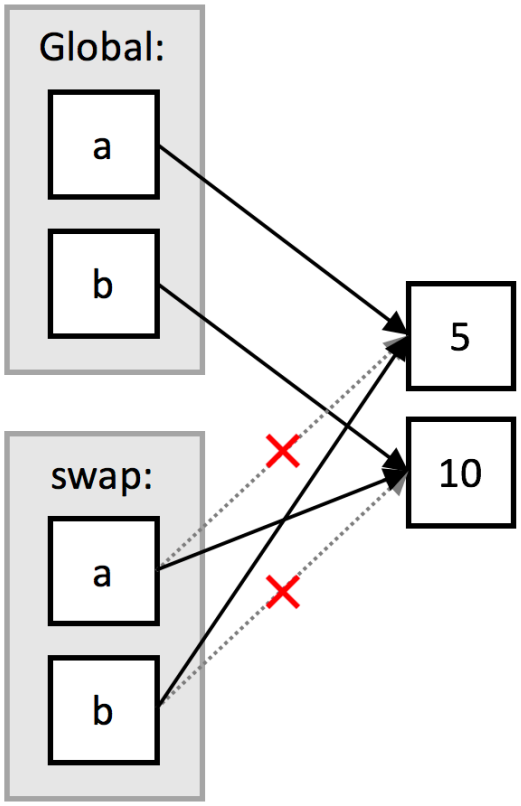
Checking for Aliases

```
## check whether two objects are exactly the same
a = [1, 2, 3]
b = a

if a is b:
    print('a and b are the same!')

b = [1, 2, 3]
if a is not b:
    print('a and b are not the same!')

if a == b:
    print('a and b are equal!')
```

- `is` and `is not` check for object equality.
- `==` and `!=` check for value equality.

References and Functions

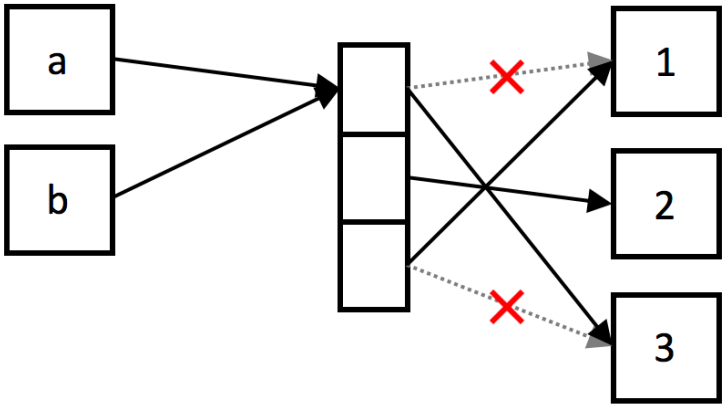
```
def swap(a, b):
    ## shorthand for swapping.
    ## the righthand side is evaluated
    ## before the left side...
    a, b = b, a
    print("in swap, a=", a)
    print("in swap, b=", b)

a = 5
b = 10
## a and b passed by reference
swap(a, b)
print("in global, a=", a)
print("in global, b=", b)
```



Global ‘a’ and ‘b’ are passed by reference into `swap`, and changes are not visible outside function.

Mutable and Immutable Types

```
## a and b are aliases to
## a mutable value!
a = [1, 2, 3]
b = a

## therefore, changes to either
## update the other
a[0] = 3
b[2] = 1
print(a)
print(b)
```



Immutable: *int*, *bool*, *float*, *str*, *tuple*, *frozenset*

Mutable: *list, dict, set*

Python Memory Model

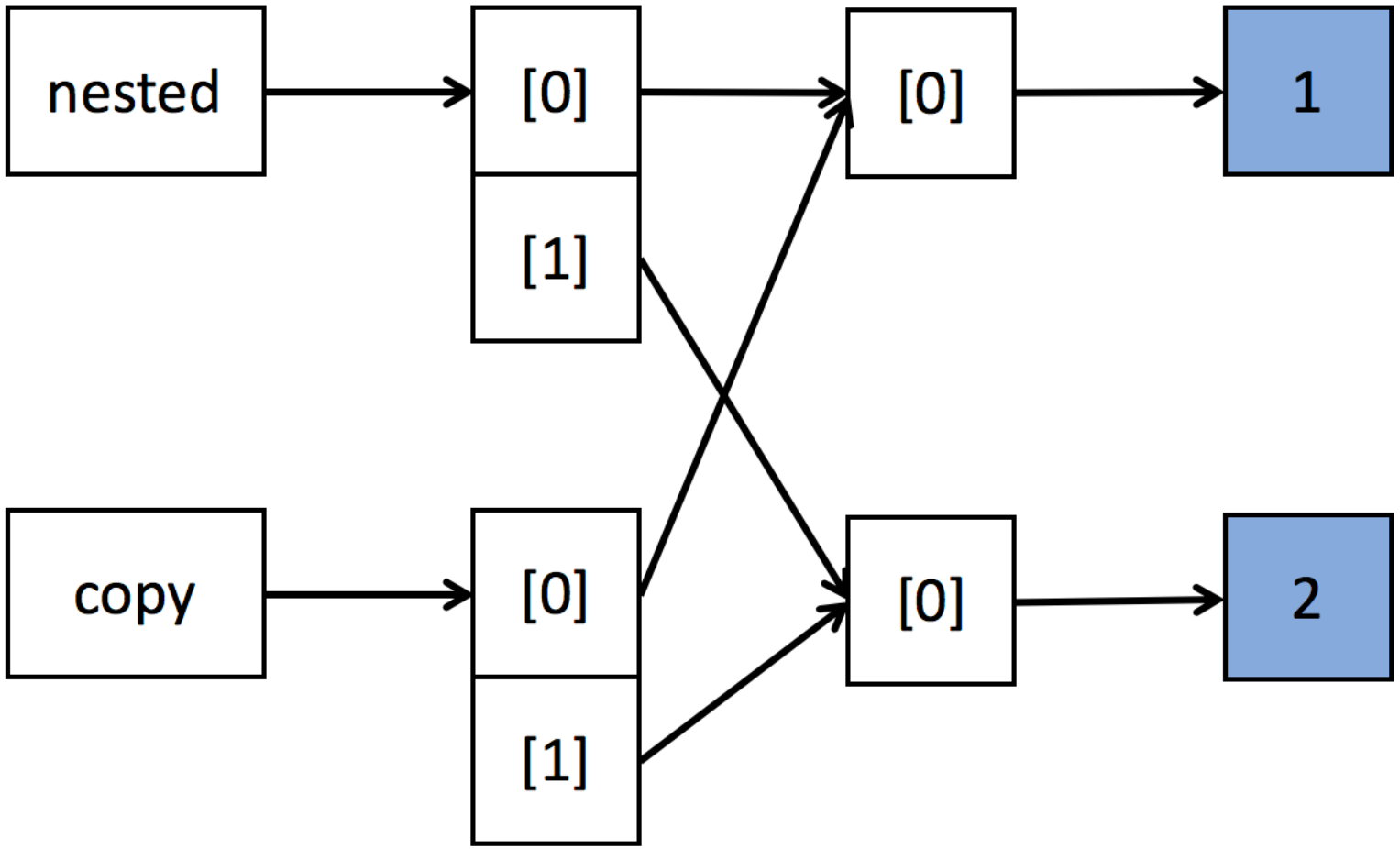
Containers and Classes

List Copies

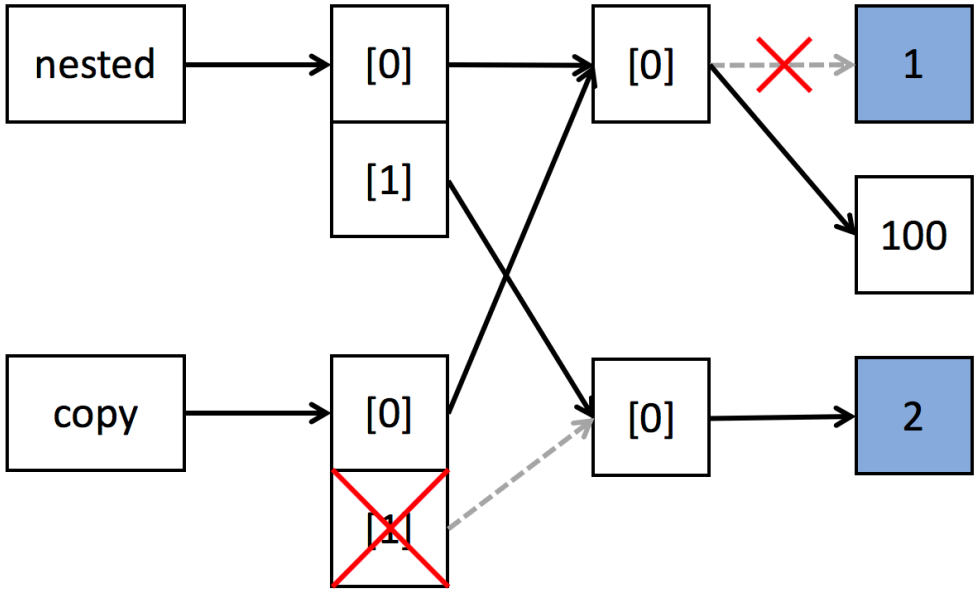
```
## create a list of two lists
nested = [ [1], [2] ]

## create a "shallow copy" of that list using
## either list(nested) or nested[:]
copy1 = nested[:]
copy2 = list(nested)

## note: these are new lists, which contain
##   references to the same nested elements!
```

- Here, we have a list containing mutable elements (in this case, other lists).
- `list()` and slice operator make "shallow" copies.

List Copies [2]

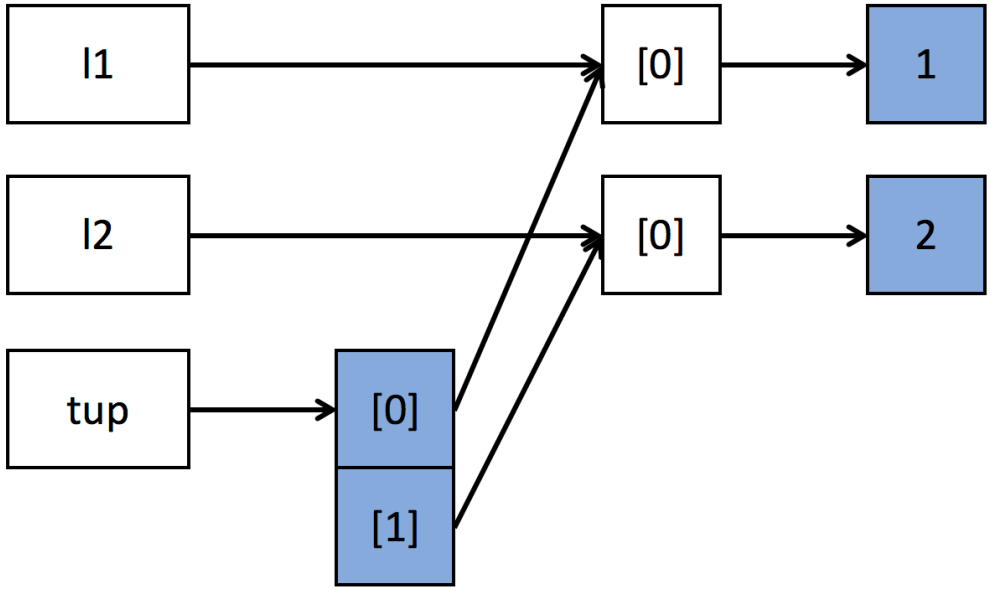


List Copies [3]



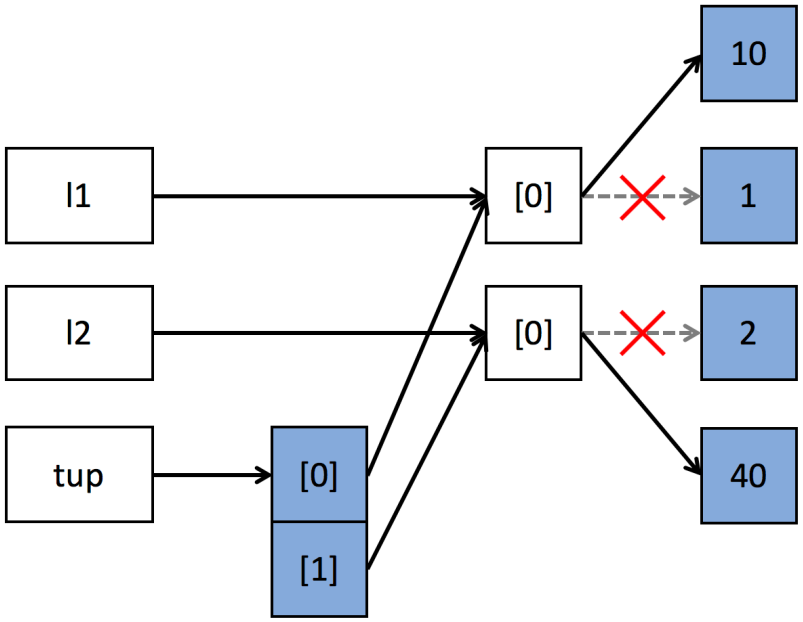

```
## changing 'copy' itself doesn't change nested
del copy[1]

## but changing an element of 'copy' may change
## 'nested' if that element is shared!
copy[0][0] = 100
```

Tuples With Mutable Contents




```
## create two lists and put them in a tuple
li1 = [1]
li2 = [2]
tup = (li1, li2)

## note: 'tup' is immutable, so its arrows
## can't change, but they point to mutable objs
```

Tuples With Mutable Contents [2]



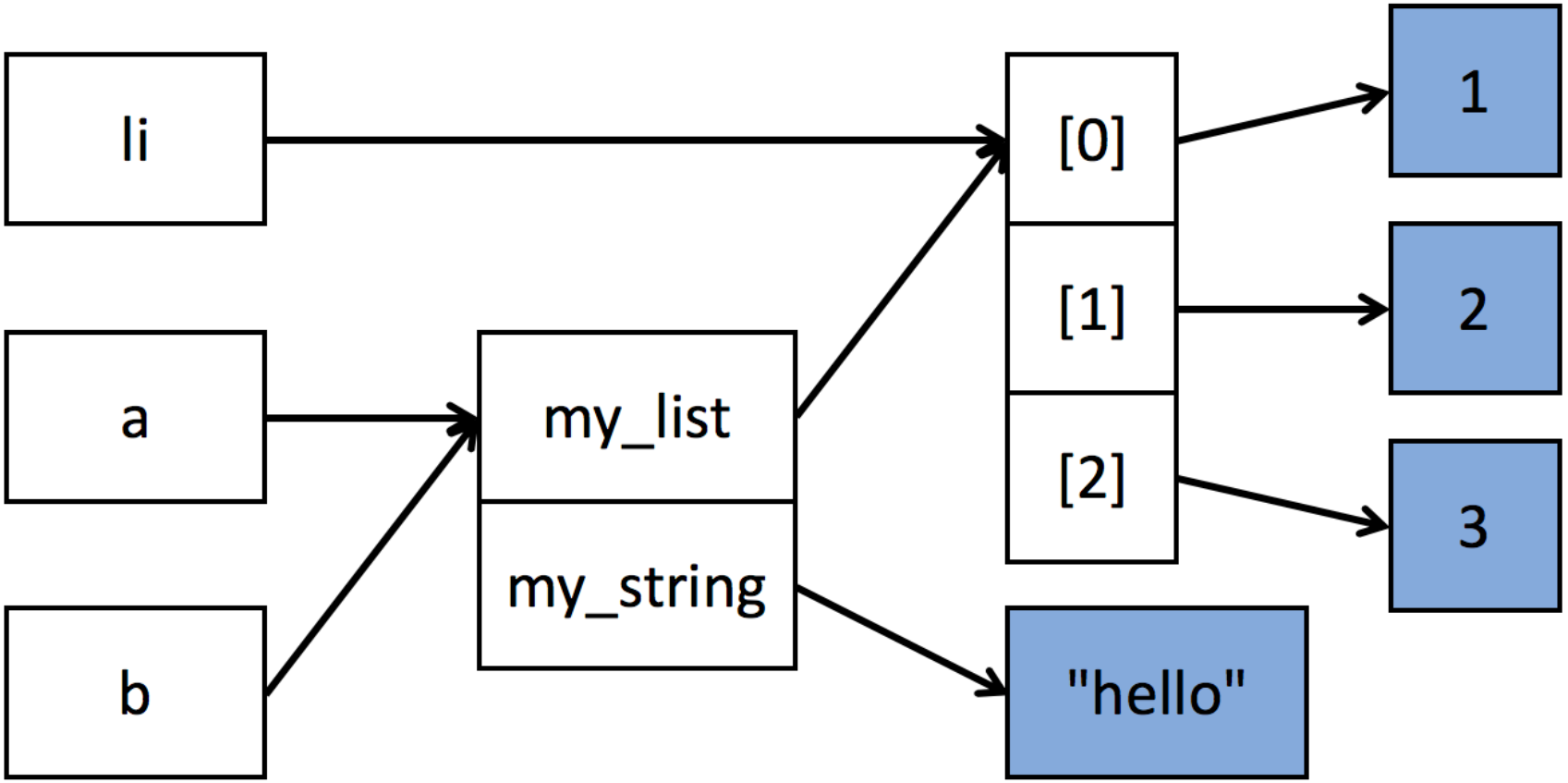

```
## we can change 'li1' and 'li2', but those are
##   aliases for tup[0] and tup[1], so changes
##   to them affect 'tup'
li1[0] = 10
li2[0] = 40
```

Classes and Mutability

```
## a custom class of a list and a string
class MyClass:
    def __init__(self, my_list, my_string):
        self.my_list = my_list
        self.my_string = my_string

## a custom class, and an alias
li = [1, 2, 3]
a = MyClass(li, "hello")
b = a
```

- By default, classes are mutable, and attributes are references to other objects.
- The code above has many aliases of mutable objects.

Classes and Mutability [2]



Shallow Copy vs. Deep Copy

Different Levels of Copying

Important programming decisions around how much copying to do

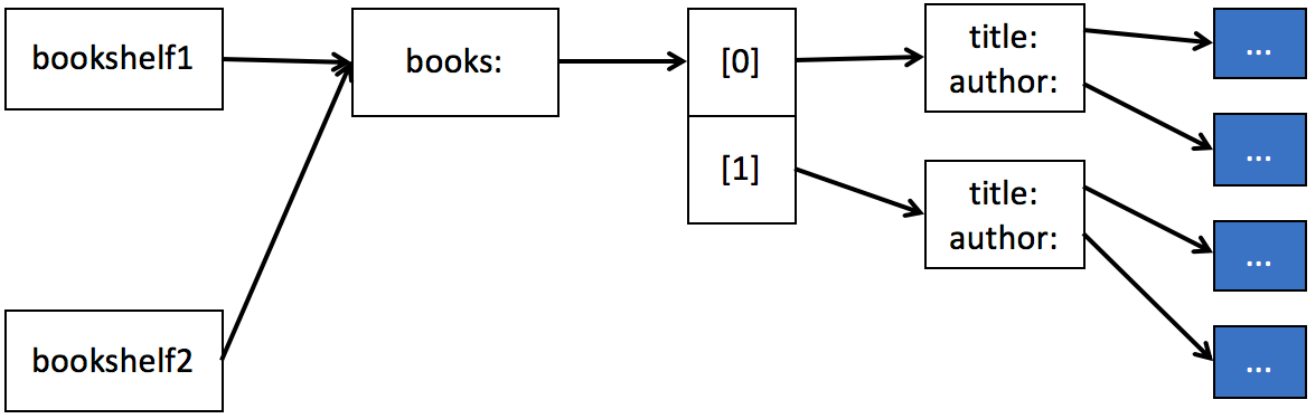
- **Aliasing:** different variables reference the same objects (e.g., = operator)
- **Shallow copy:** creates a new object at the top level, but contents are still aliased (e.g., `list(…)` built-in function)
- **Deep copy:** new objects are created all the way down, no shared objects (e.g., `copy` module)

Example: Bookshelf Class

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

class Bookshelf:
    def __init__(self, books):
        ## note: implementation changes below
        ## this implementation is making an alias
        self.books = books
```

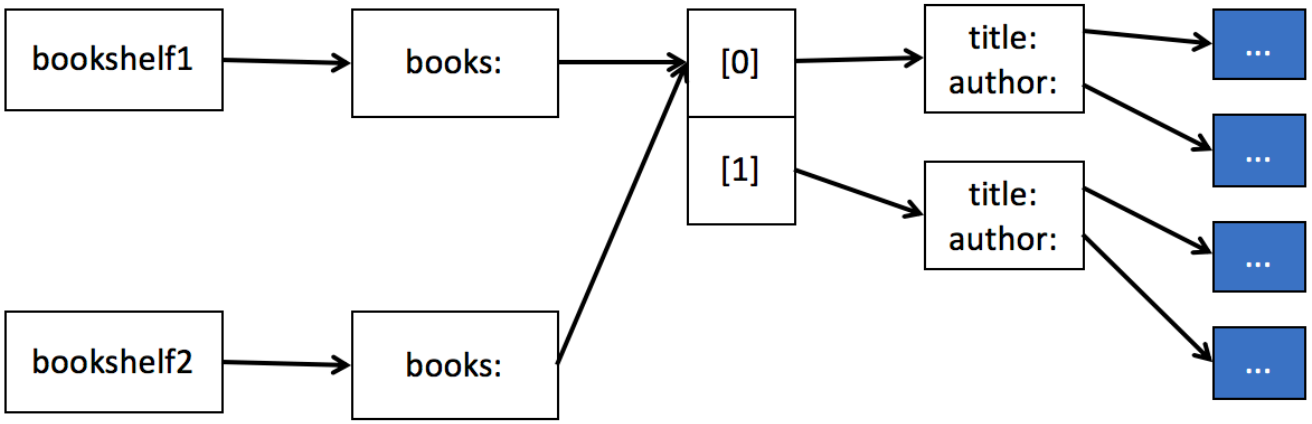
- Simple class definition of `Book` with a constructor and two attributes
- `Bookshelf` has a single attribute, books
- The Bookshelf constructor will be changed to reflect different approaches to copying

Approach 1: Alias to Bookshelf Object



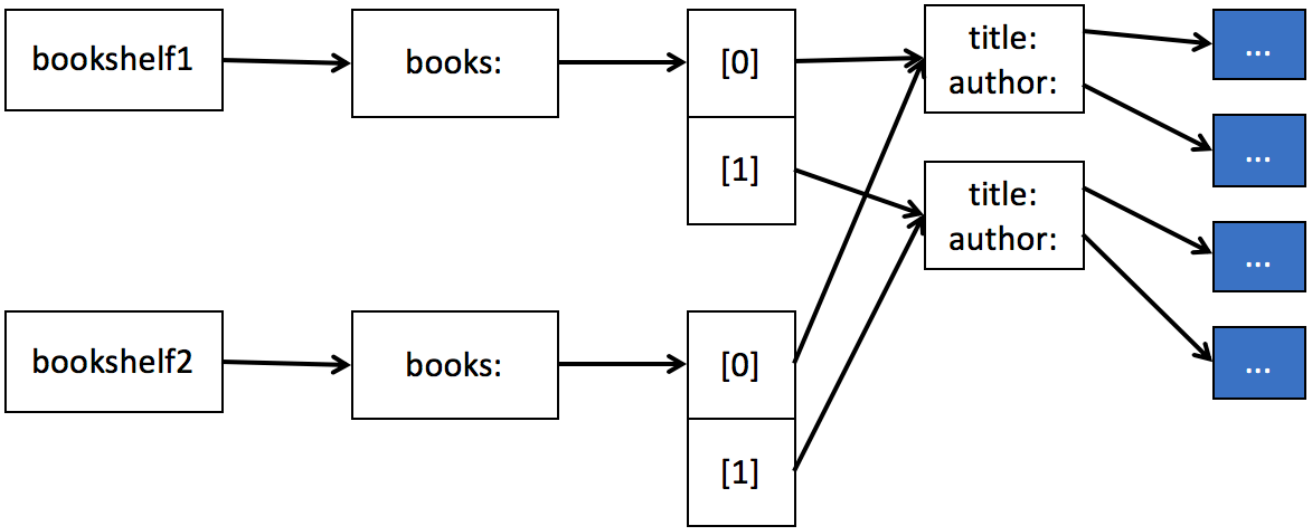

```
books = [
    Book('It', 'Stephen King'),
    Book('American Gods', 'Neil Gaiman') ]
bookshelf1 = Bookshelf(books)
bookshelf2 = bookshelf1
## note: aliasing → two names for the same object
```

Approach 2: Shallow Copy of Bookshelf




```
books = [
    Book('It', 'Stephen King'),
    Book('American Gods', 'Neil Gaiman') ]
bookshelf1 = Bookshelf(books)
bookshelf2 = Bookshelf(books)
```

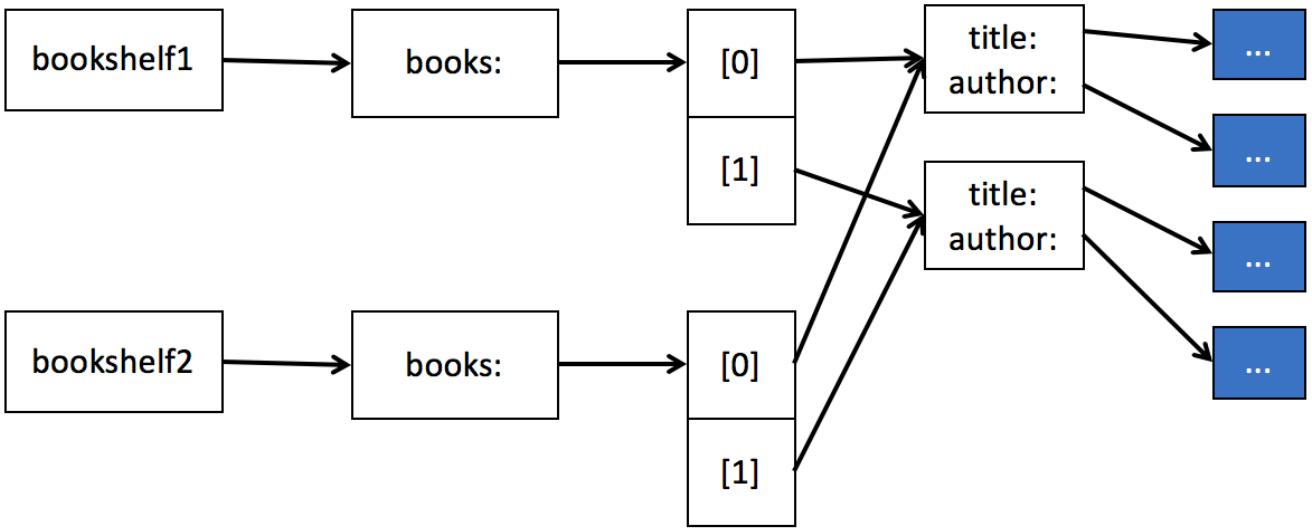
Approach 3: Shallow Copy of Bookshelf.Books




```
class Bookshelf:
    def __init__(self, books):
        self.books = books[:]
```

Enhanced implementation of the Bookshelf constructor forces a shallow copy.

Approach 4: Deep Copy of Bookshelf.Books




```
class Bookshelf:
    def __init__(self, books):
        self.books = [ Book(b.title, b.author) for b in
        books ]
```

Enhanced implementation of the Bookshelf constructor contains an explicit deep copy.

Value Equality vs. Reference Equality

Reference Equality

- Occurs as a result of aliasing
- Two variables reference the same object
- Testable with the `is` and `is not` operators
- Very efficient (just compares addresses)
- Internally, the `id(obj)` function is used

Reference Equality [2]

```
## three variables: two of which are aliases
a = [1, 2, 3]
b = [1, 2, 3]
c = a

## 'a' and 'b' are not reference equal
## because they are not aliases
assert a is not b
assert b is not c

## 'a' and 'c' are reference equal
## because they are aliases
assert a is c
```

Value Equality

- More standard definition of “equality”
- Two variables are equal if they represent the same value
- Testable with == and != operators
- Can be computationally costly with large collections and deep object hierarchies

Value Equality [2]

```
## three variables: two of which are aliases
a = [1, 2, 3]
b = [1, 2, 3]
c = a

## each variable is value equal since they
## are all lists containing the values 1, 2, and 3
assert a == b
assert a == c
assert b == c
assert a is c
assert b is not c
```

The common equality operators, == and !=, use value equality.