# Lists

***Special Topic in Computer Science***
***Python Programming***
*Dr. Kamel Rushaidat*

# Objectives

- To understand Python lists
- To use lists as a means of storing data
- To use sets mathematical operators.
- To use dictionaries to store associative data
- To implement algorithms to compute elementary statistics

# What Is Data?

- *Data* (raw data) is an assortment of items that have been observed, measured, or collected by some means.
- Represents the starting point for analysis that can be done in an attempt to discover underlying characteristics that might be present and are typically referred to as *information*.
- This analysis is based on the mathematical science of *statistics*.

# Storing Data for Processing

- Anytime that we work with large amounts of data it is necessary to have some means of organized storage so that processing of the data can take place in an orderly and efficient manner.

# Data Structures

- Data structures are particular ways of storing data to make some operation easier or more efficient. That is, they are tuned for certain tasks
- Data structures are suited to solving certain problems, and they are often associated with algorithms.

# Kinds of data structures

Roughly two kinds of data structures:

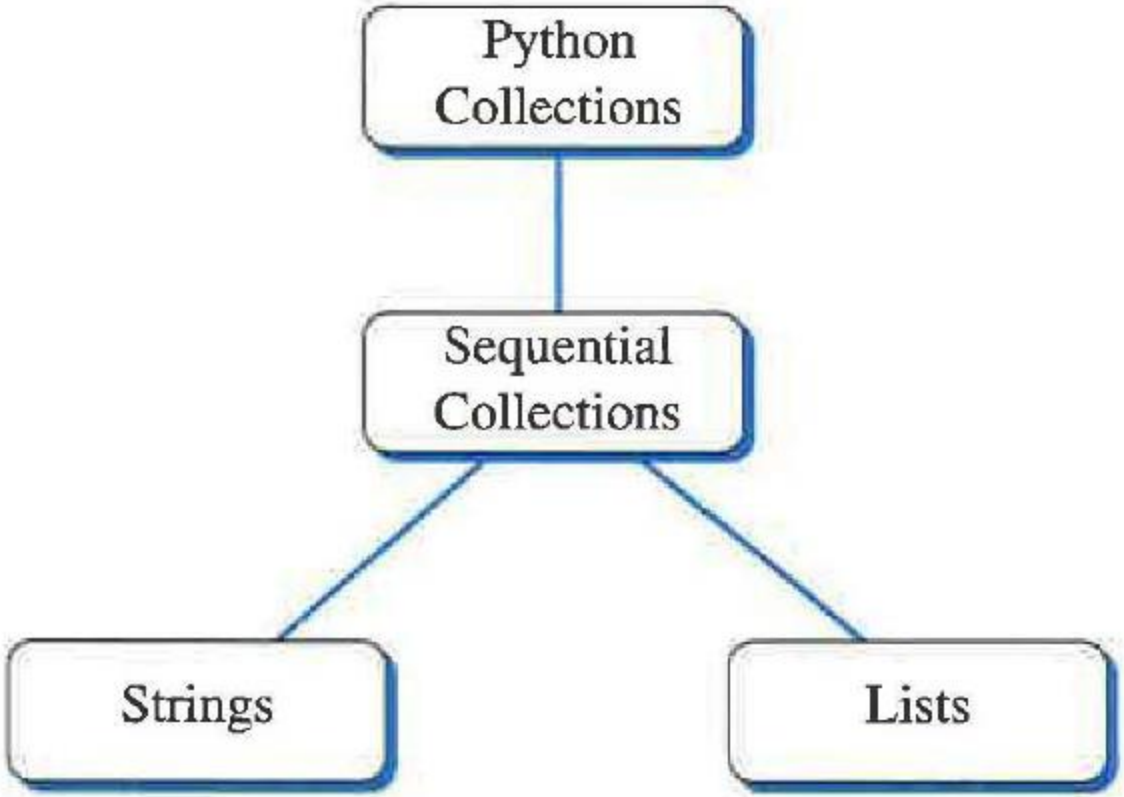
- built-in data structures, data structures that are so common as to be provided by default
- user-defined data structures (classes in object oriented programming) that are designed for a particular task

# Python built in data structures

- Python comes with a general set of built in data structures:
  - lists
  - tuples
  - string
  - dictionaries
  - sets
  - others...

# Lists and Strings as Sequential Collections

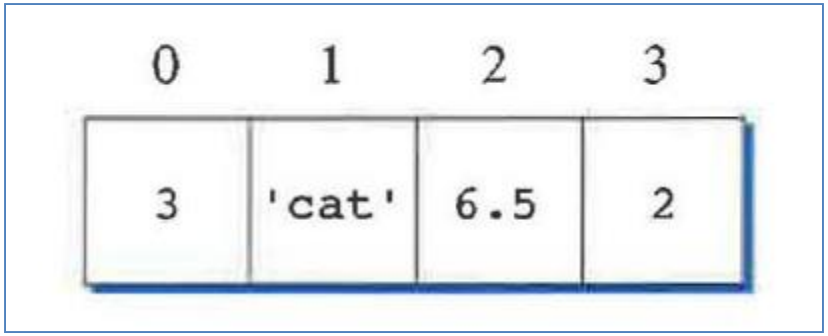
We will first consider two of Python's built-in collections as a means of storing our data values: strings and lists.



# Lists

- A list is an ordered, sequential collection of zero or more Python data objects.
- Lists are written as comma-delimited values enclosed in square brackets.
- We call a list with zero data objects the *empty list*, which is represented simply by [ ].
- Strings are *homogeneous* collections because each item in the collection is the same type of object-a character. Lists, by contrast, are *heterogeneous* and can be composed of any kind of object.

# Lists (cont'd.)




```
>>> [3,"cat",6.5,2]
[3, 'cat', 6.5, 2]
>>> mylist = [3,"cat",6.5,2]
>>> mylist
[3, 'cat', 6.5, 2]
>>>
```

# Using Sequence Operators with Lists

| Operation Name | Operator | Explanation |
|---|---|---|
| Indexing | [] | Access an element of a sequence |
| Concatenation | + | Combine sequences together |
| Repetition | * | Concatenate a repeated number of times |
| Membership | in | Ask whether an item is in a sequence |
| Membership | not in | Ask whether an item is not in a sequence |
| Length | len | Ask the number of items in the sequence |
| Slicing | [:] | Extract a part of a sequence |

- Since lists are considered to be sequential, they support a number of operations that can be applied to any Python sequence.
- These are the same operations that we used with strings since both are composed of a sequential collection of items.

# Using Sequence Operators with Lists (cont'd.)

```
>>> mylist
[1, 3, 'cat', 4.5]
>>> mylist[2]
'cat'
>>> mylist+mylist
[1, 3, 'cat', 4.5, 1, 3, 'cat', 4.5]
>>> mylist*3
[1, 3, 'cat', 4.5, 1, 3, 'cat', 4.5, 1, 3, 'cat', 4.5]
>>> len(mylist)
4
>>> len(mylist*4)
16
>>> mylist[1:3]
[3, 'cat']
>>> 3 in mylist
True
>>> "dog" in mylist
False
>>> del myList[2]
>>> myList
[1, 3, 4.5]
```

# Using Sequence Operators with Lists (cont'd.)

- Indices for lists, *as with strings, start with 0.*
- *The slice operation mylist [1: 3]* returns a list of items starting with the item indexed by 1 up to but not including the item indexed by 3.

# Using the List Function

- A useful function for creating lists is the *list* function.
- The list function converts other sequences to lists.
- We have seen two such sequences: strings and ranges.

```
>>> range(10)
range(10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(10,2,-2))
[10, 8, 6, 4]
>>> list(''the quick fox'')
['t', 'h', 'e', ' ', 'q', 'u', 'i', 'c', 'k', ' ', 'f', 'o', 'x']
>>>
```

# Methods Provided by Lists in Python

| Method Name | Use | Explanation |
|---|---|---|
| append | alist.append(item) | Adds a new item to the end of a list |
| insert | alist.insert(i,item) | Inserts an item at the ith position in a list |
| pop | alist.pop() | Removes and returns the last item in a list |
| pop | alist.pop(i) | Removes and returns the ith item in a list |
| sort | alist.sort() | Modifies a list to be sorted |
| reverse | alist.reverse() | Modifies a list to be in reverse order |
| index | alist.index(item) | Returns the index of the first occurrence of item |
| count | alist.count(item) | Returns the number of occurrences of item |
| remove | alist.remove(item) | Removes the first occurrence of item |

# Methods Provided by Lists in Python (cont'd.)

```
>>> mylist
[1024, 3, True, 6.5]
>>> mylist.append(False)
>>> mylist
[1024, 3, True, 6.5, False]
>>> mylist.insert(2,4.5)
>>> mylist
[1024, 3, 4.5, True, 6.5, False]
>>> mylist.pop()
False
>>> mylist
[1024, 3, 4.5, True, 6.5]
>>> mylist.pop(1)
3
>>> mylist
[1024, 4.5, True, 6.5]
>>> mylist.pop(2)
True
```

```
>>> mylist
[1024, 4.5, 6.5]
>>> mylist.sort()
>>> mylist
[4.5, 6.5, 1024]
>>> mylist.reverse()
>>> mylist
[1024, 6.5, 4.5]
>>> mylist.count(6.5)
1
>>> mylist.index(4.5)
2
>>> mylist.remove(6.5)
>>> mylist
[1024, 4.5]
>>>
```

# Using the Split Method

- *split* takes a string *as a parameter that indicates the places to break the string into* substrings.
- The substrings are returned in a list.
- By default, if no parameter is passed to split, it will break the string using one or more spaces *as the delimiter.*

```
>>> a = "minnesota vikings"
>>> a.split()
['minnesota', 'vikings']
>>> a.split('i')
['m', 'nnesota v', 'k', 'ngs']
>>> a.split('nn')
['mi', 'esota vikings']
>>>
```

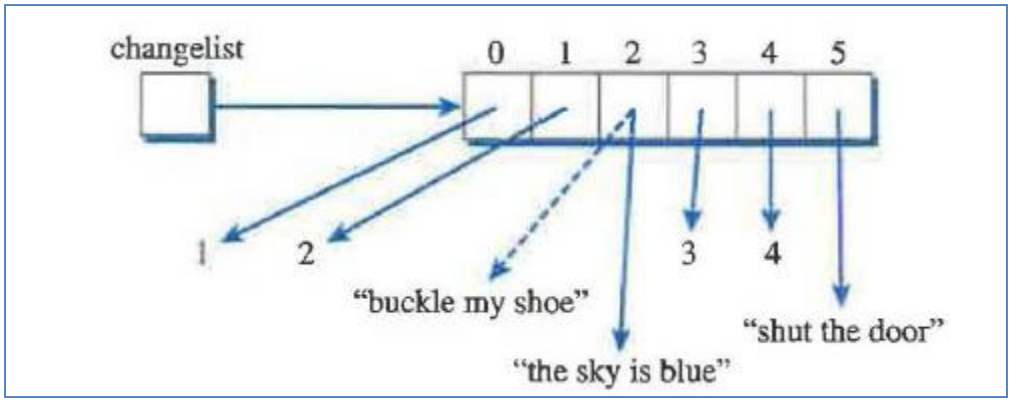
```
>>> str = "United States Of America"
>>> word1,word2,word3,word4 = str.split()
>>> word1
'United'
>>> word2
'States'
>>> word3
'Of'
>>> word4
'America'
>>> str2 = "Jack,Tom,John"
>>> name1,name2,name3 = str2.split(',')
>>> name1
'Jack'
>>> name2
'Tom'
>>> name3
'John'
```

# Differences between Lists and Strings

- lists can contain a mixture of any python object, strings can only hold characters
  - 1,"bill",1.2345, True
- lists are mutable, their values can be changed, while strings are immutable
- lists are designated with [ ], with elements separated by commas, strings use " " or ' '

# Mutating a List

- Strings are *immutable* collections of data where individual items within the string cannot be changed, however, lists are *mutable* collections of data that can be modified.



# Mutating a List (cont'd.)

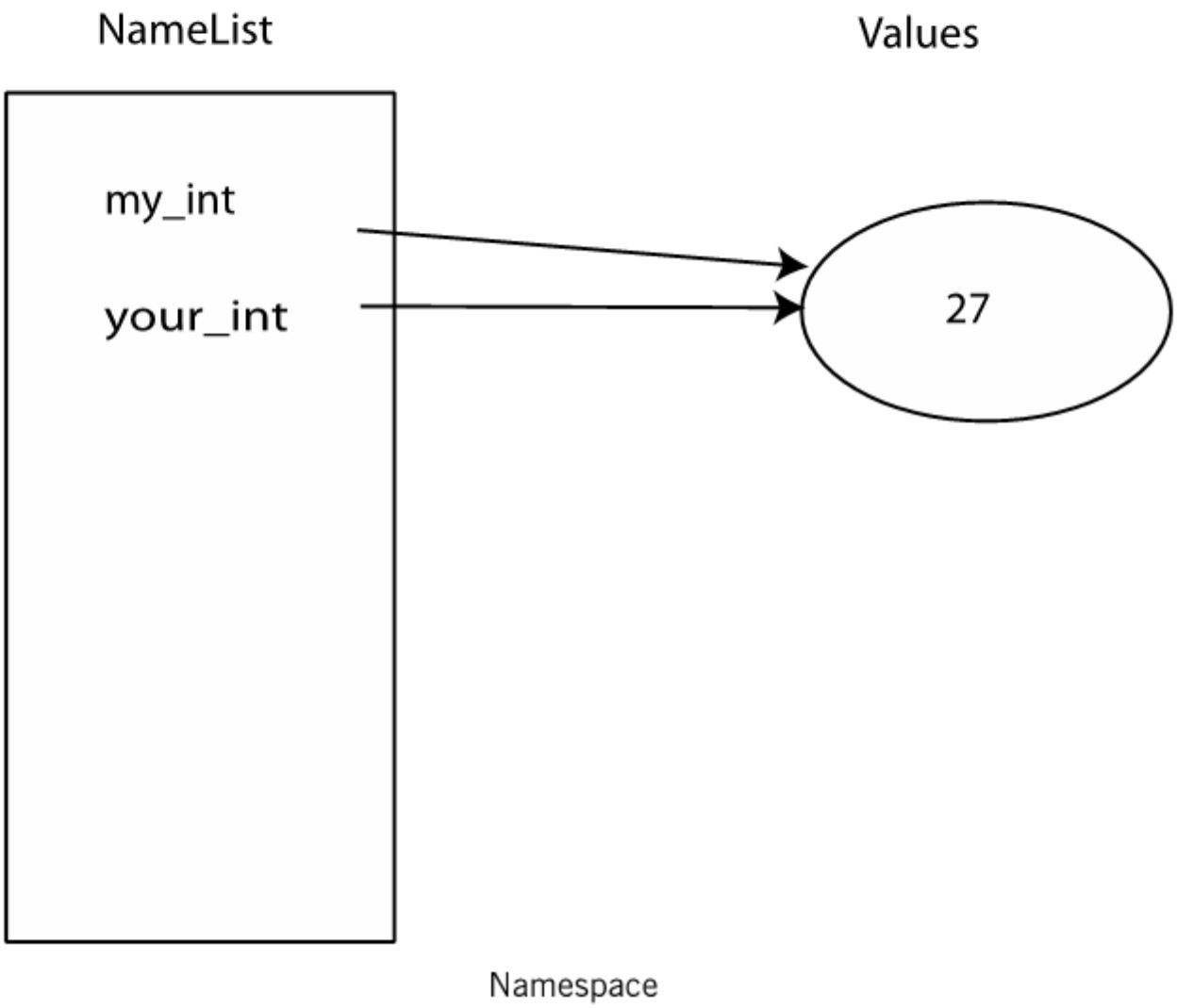
```
>>> changelist = [1,2,"buckle my shoe",3,4,"shut the door"]
>>> changelist
[1, 2, 'buckle my shoe', 3, 4, 'shut the door']
>>> changelist[2] = "the sky is blue"
>>> changelist
[1, 2, 'the sky is blue', 3, 4, 'shut the door']
>>>
>>> name = "Monte"
>>> name[2] = "x"

Traceback (most recent call last):
  File "<pyshell#67>", line 1, in -toplevel-
    name[2] = "x"
TypeError: object does not support item assignment
>>>
```

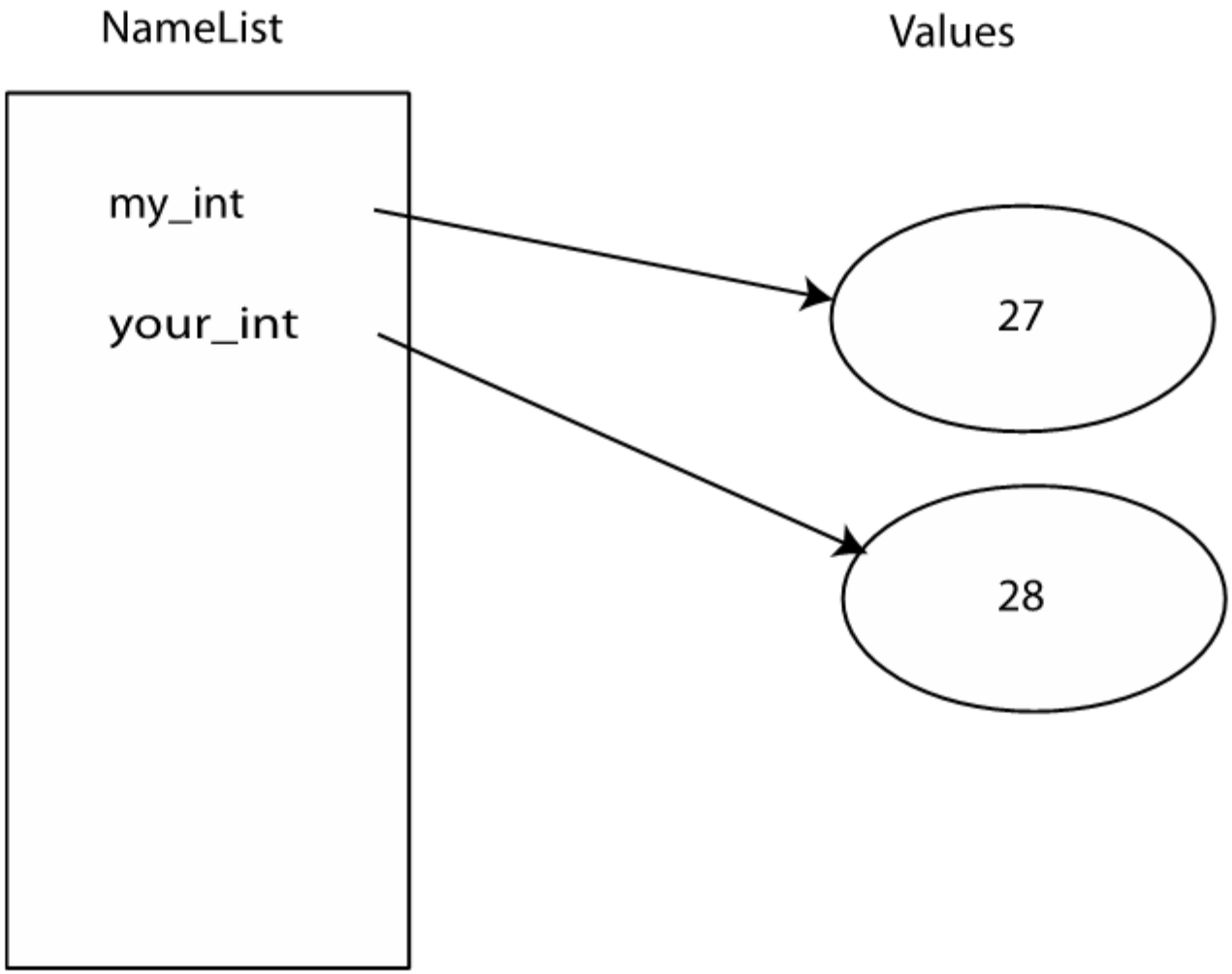
# Immutables

- Assignment takes an object (the final object after all operations) from the RHS 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
- 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.

my_int = 27
your_int = my_int



```
my_int = 27
your_int = my_int
your_int = your_int + 1
```

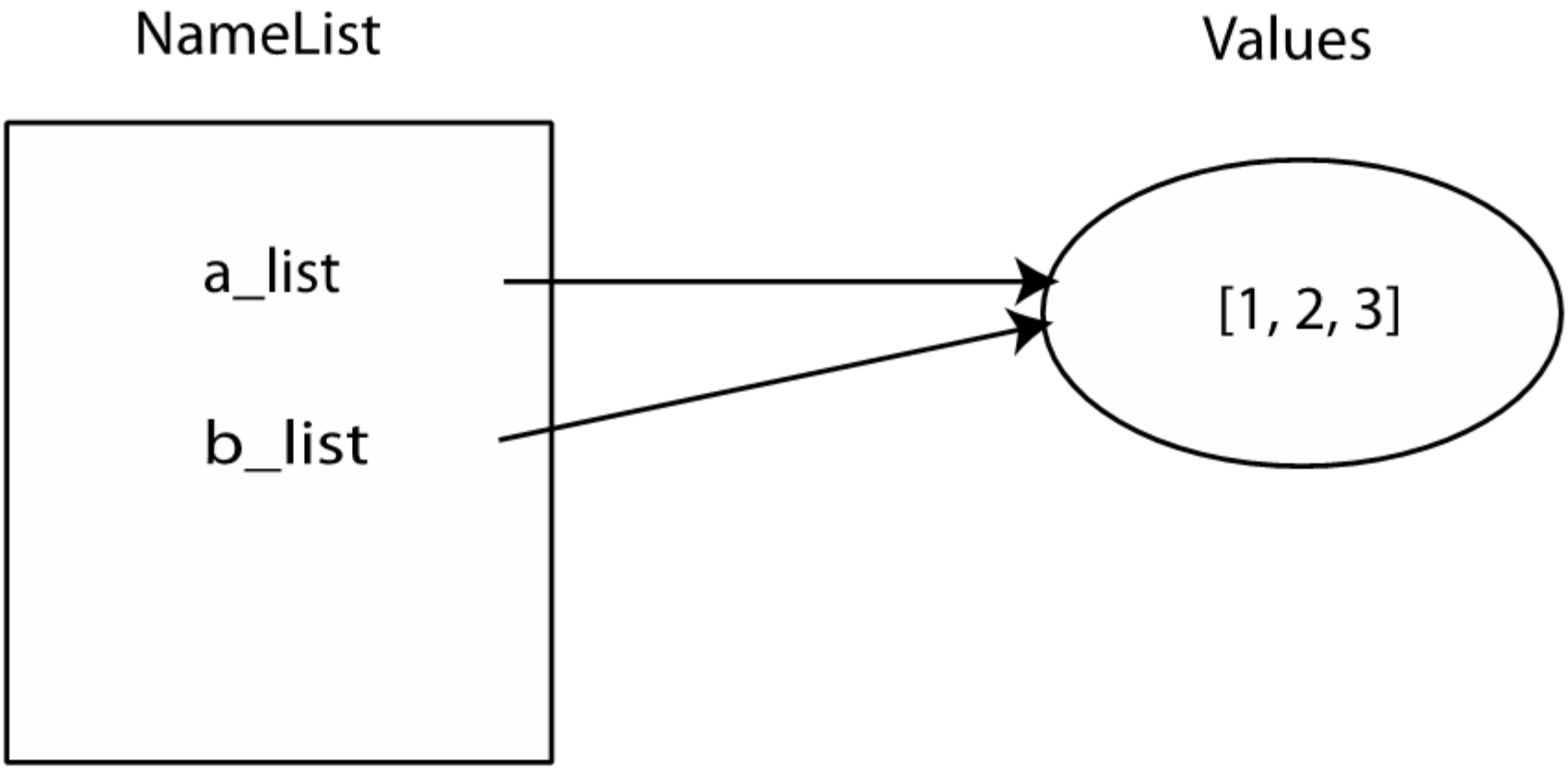



Modification of a reference to an immutable object.

# Mutability

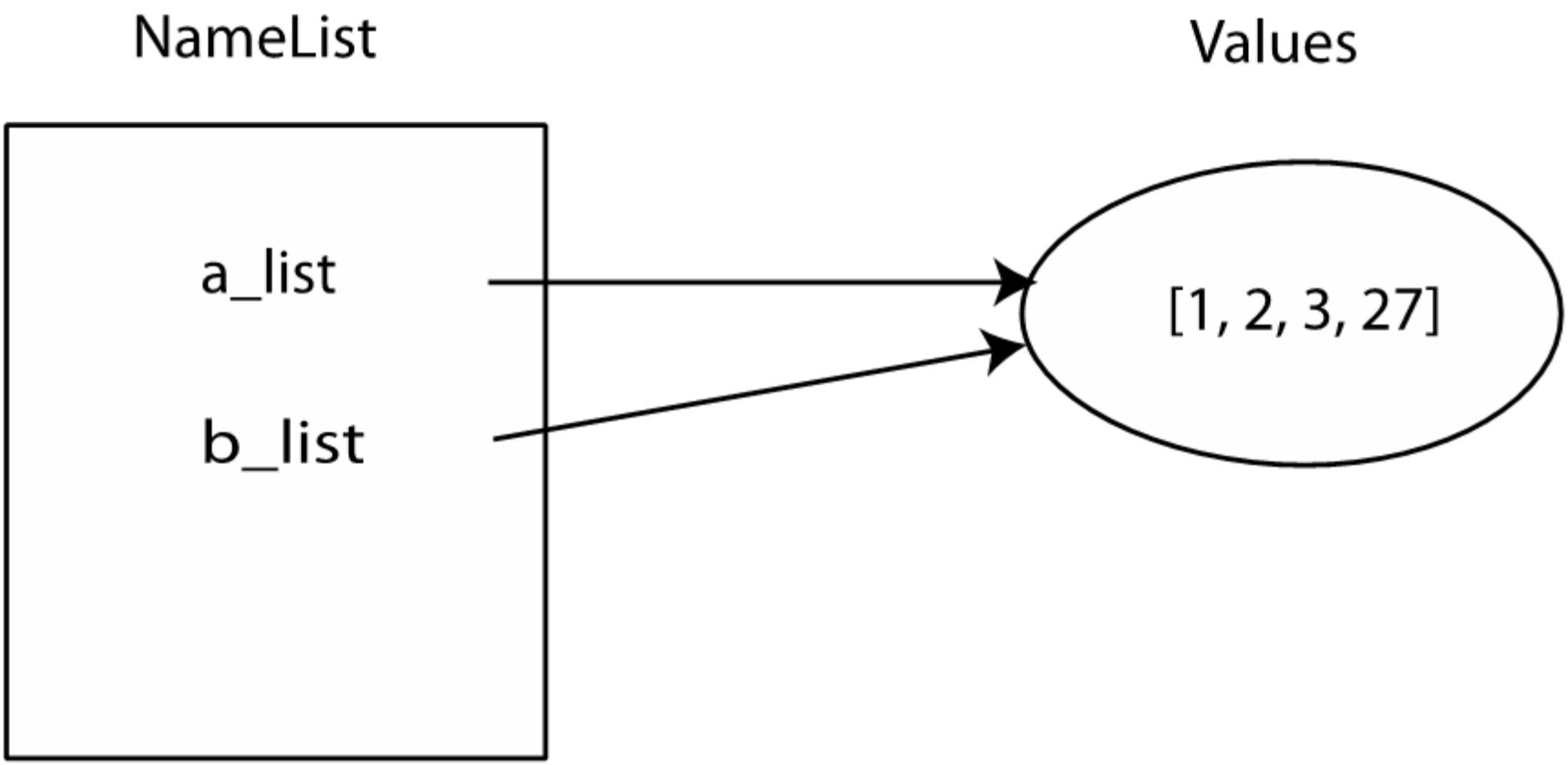
- If two variables associate with the same object, then ***both reflect*** any change to that object

```
a_list = [1,2,3]
b_list = a_list
```




Namespace snapshot after assigning mutable objects.

```
a_list = [1,2,3]
b_list = a_list
a_list.append(27)
```



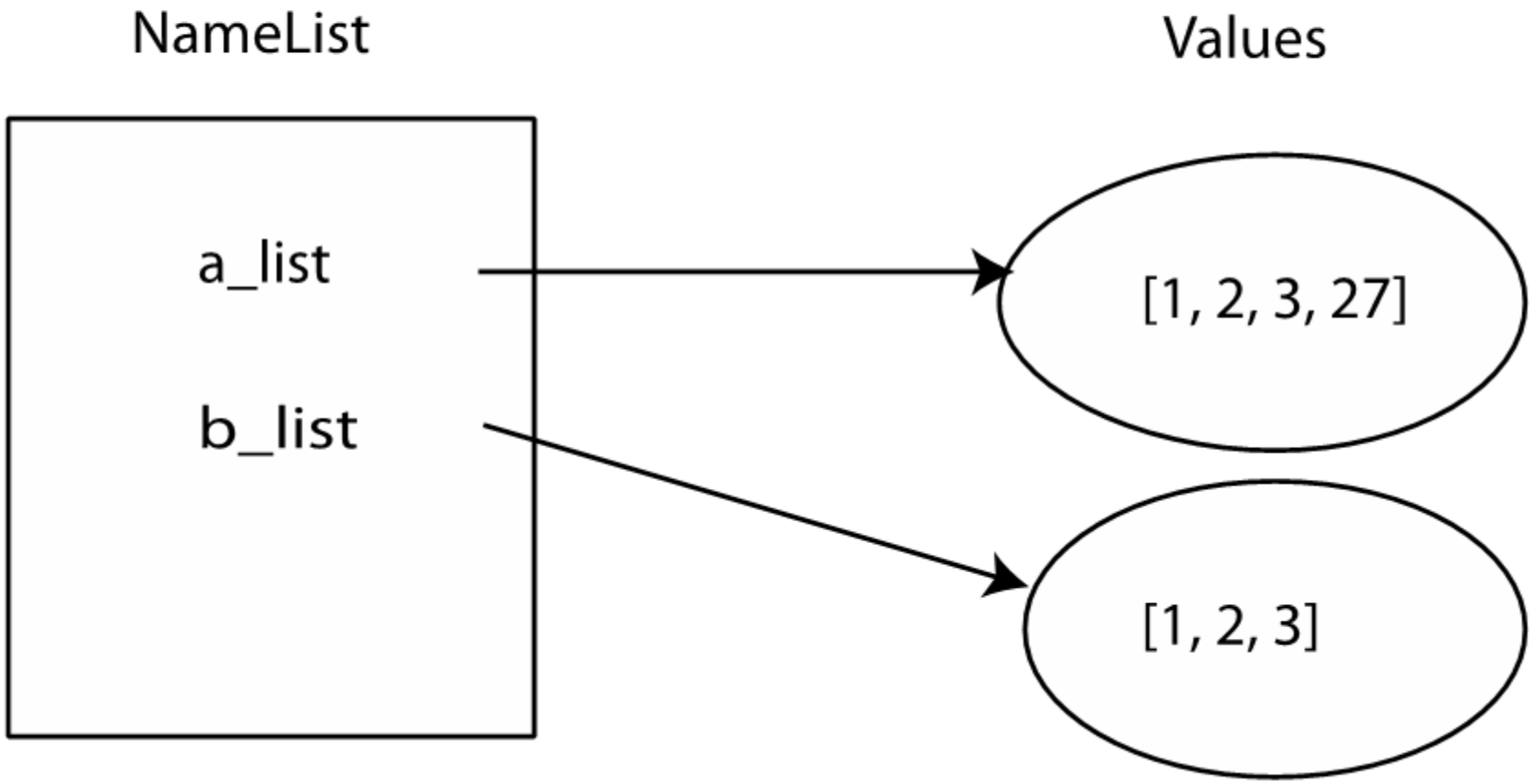

Modification of shared, mutable objects.

# Copying

If we copy, does that solve the problem?

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

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



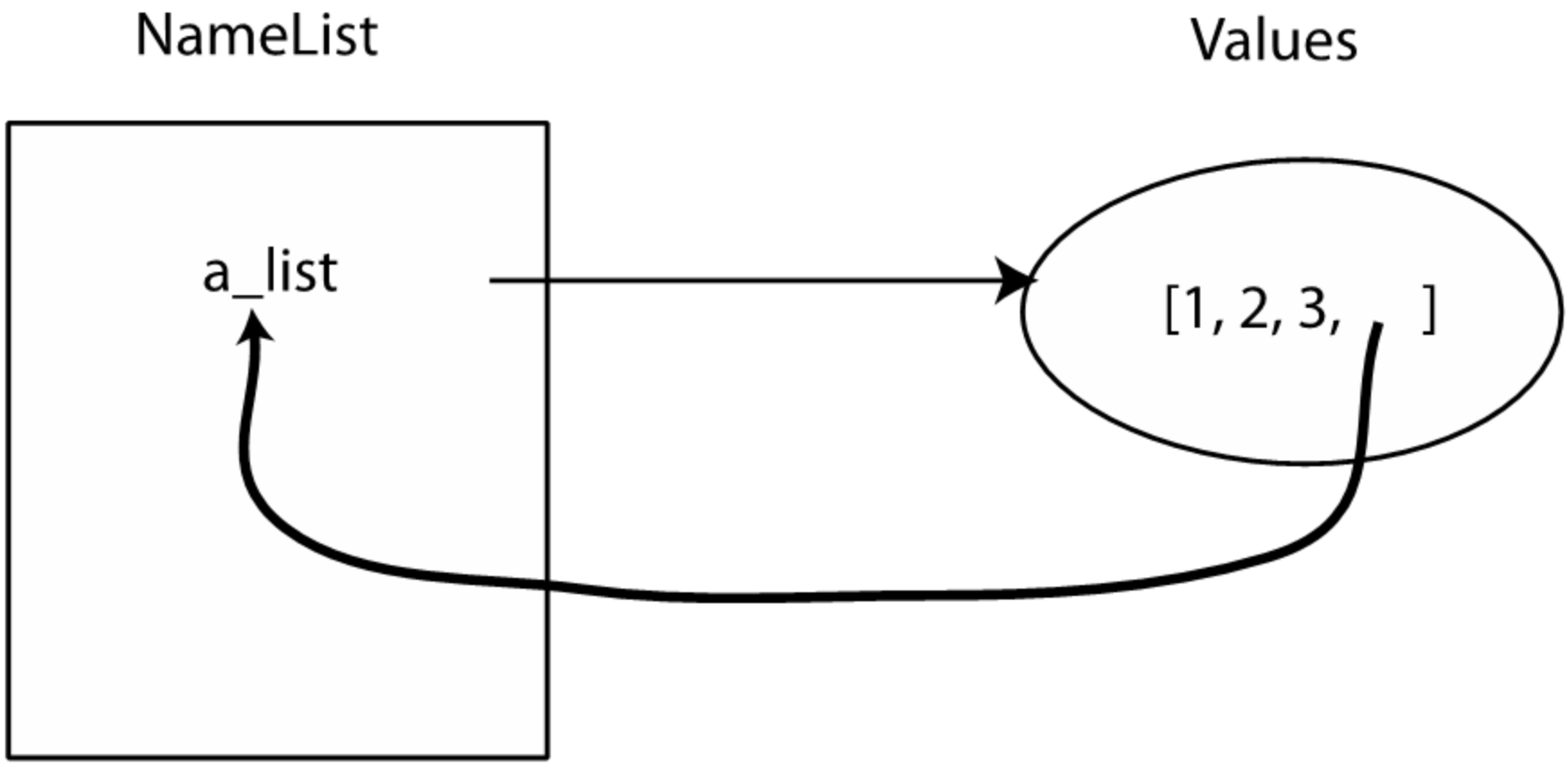

Making a distinct copy of a mutable object.

# Sort_of/depends

The big question is, what gets copied?

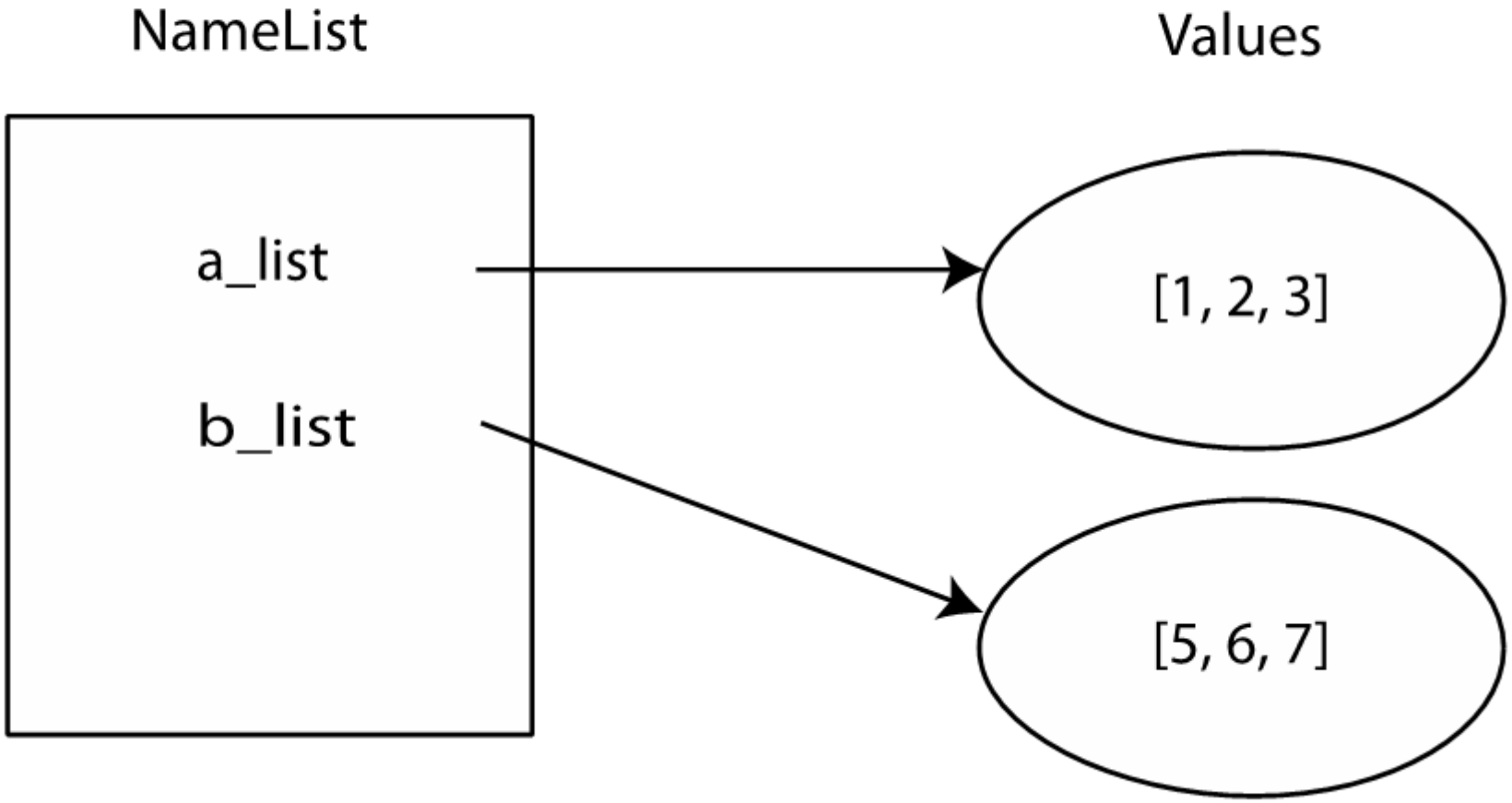
- What actually gets copied is the top level reference. If the list has nested lists or uses other associations, the association gets copied. This is termed a ***shallow copy***.

```
a_list = [1,2,3]
a_list.append(a_list)
print(a_list)  ⟶  [1, 2, 3, [...]]
```



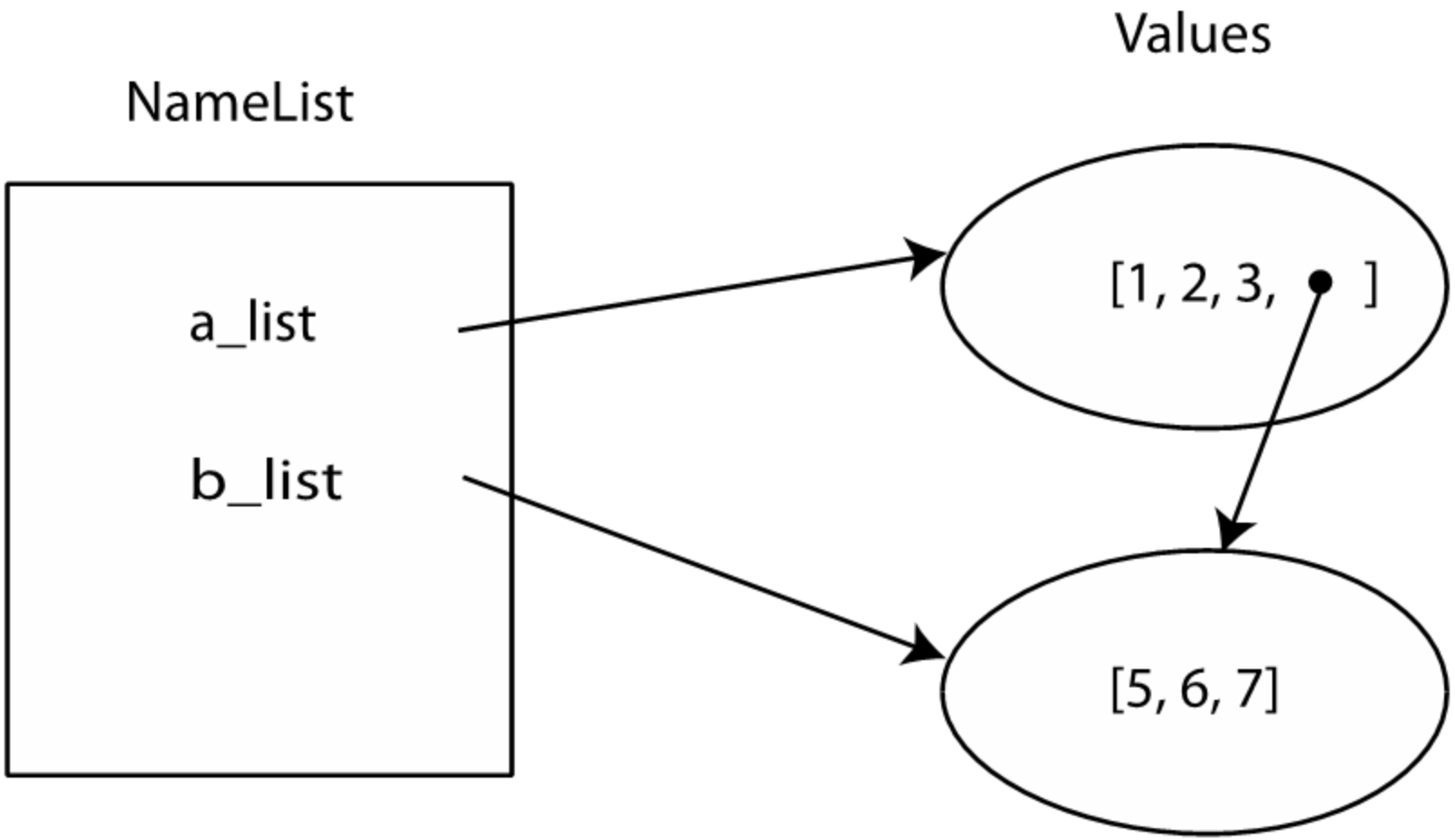

Self-referencing.

a_list = [1,2,3]
b_list = [5,6,7]



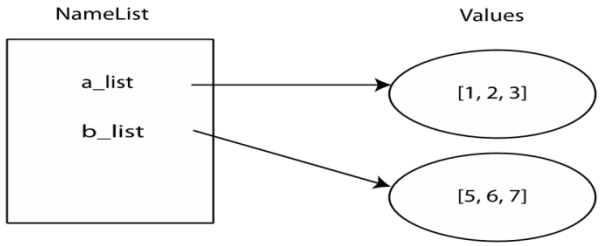

Simple lists before append.

```
a_list = [1,2,3]
b_list = [5,6,7]
a_list.append(b_list)
```

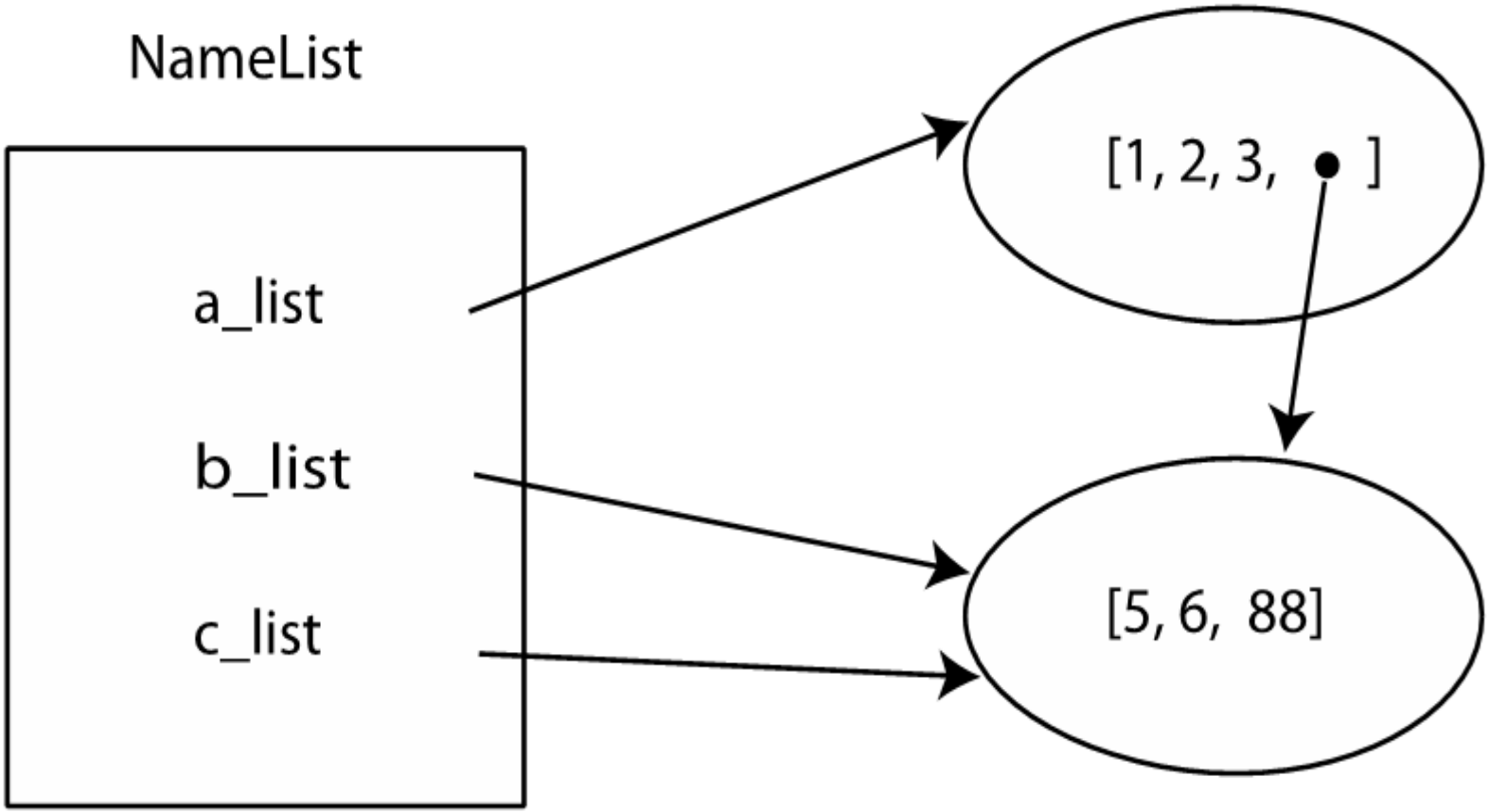



Lists after append.

```
a_list = [1,2,3]
b_list = [5,6,7]
a_list.append(b_list)
c_list = b_list
c_list[2] = 88
```




Simple lists before append.




Final state of copying example.

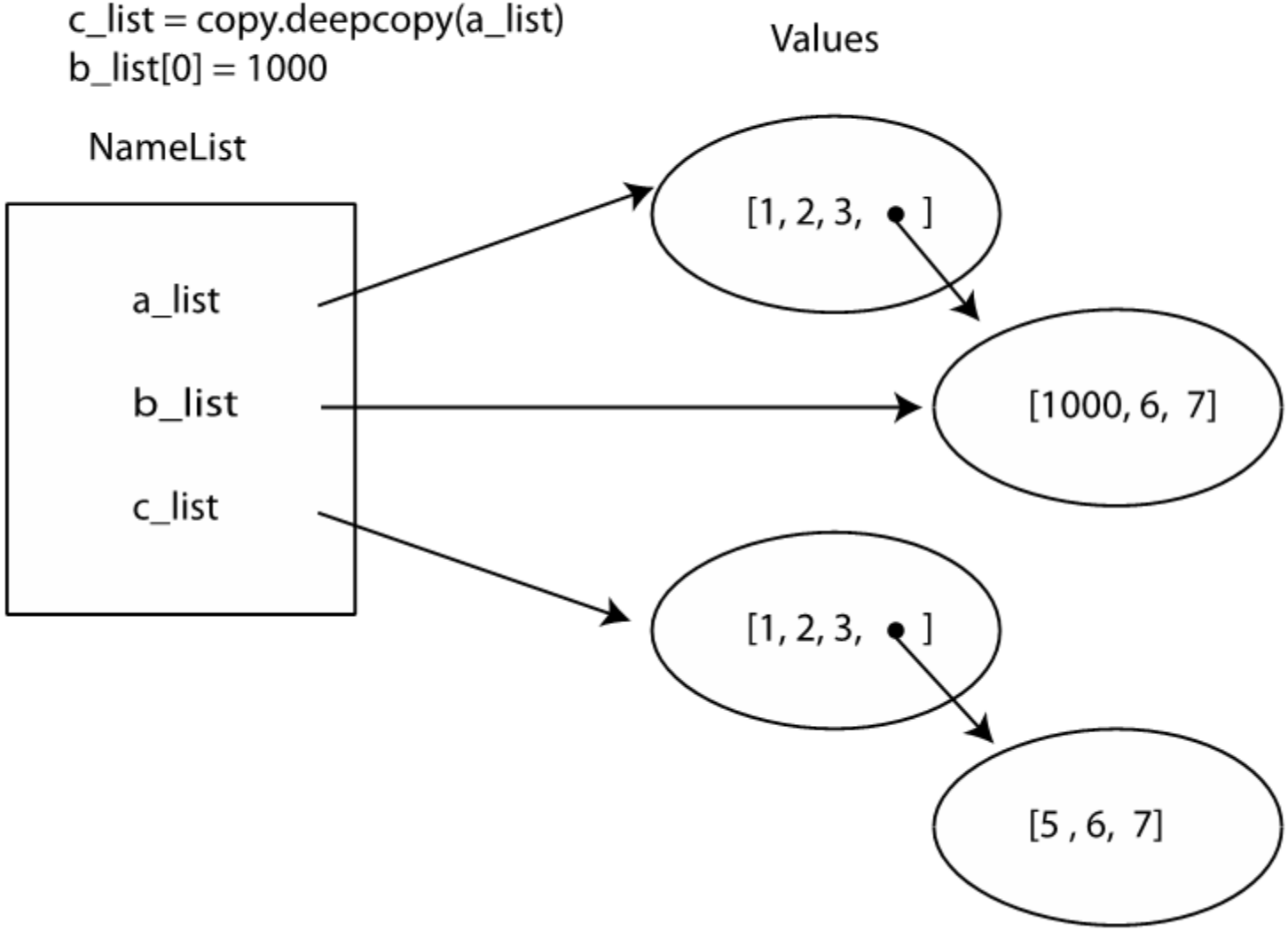
# shallow vs deep

Regular copy, the `[:]` approach, only copies the top level reference/association

- if you want a full copy, you can use deepcopy

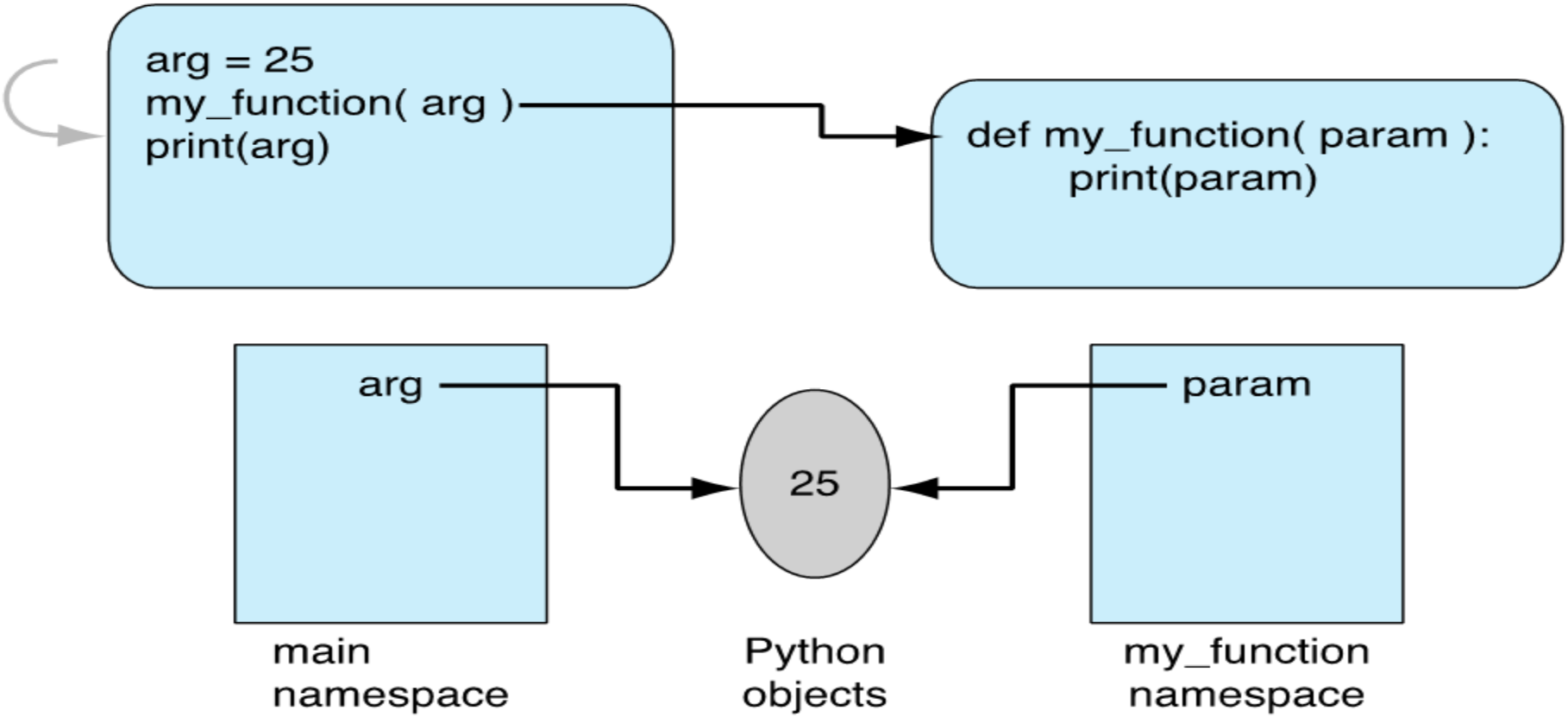
```
>>> a_list = [1, 2, 3]
>>> b_list = [5, 6, 7]
>>> a_list.append(b_list)
>>> import copy
>>> c_list = copy.deepcopy(a_list)
>>> b_list[0]=1000
>>> a_list
[1, 2, 3, [1000, 6, 7]]
>>> c_list
[1, 2, 3, [5, 6, 7]]
>>>
```

```
a_list = [1,2,3]
b_list = [5,6,7]
a_list.append(b_list)
c_list = copy.deepcopy(a_list)
b_list[0] = 1000
```




Using the `copy` module for a deep copy.

# Passing immutable objects



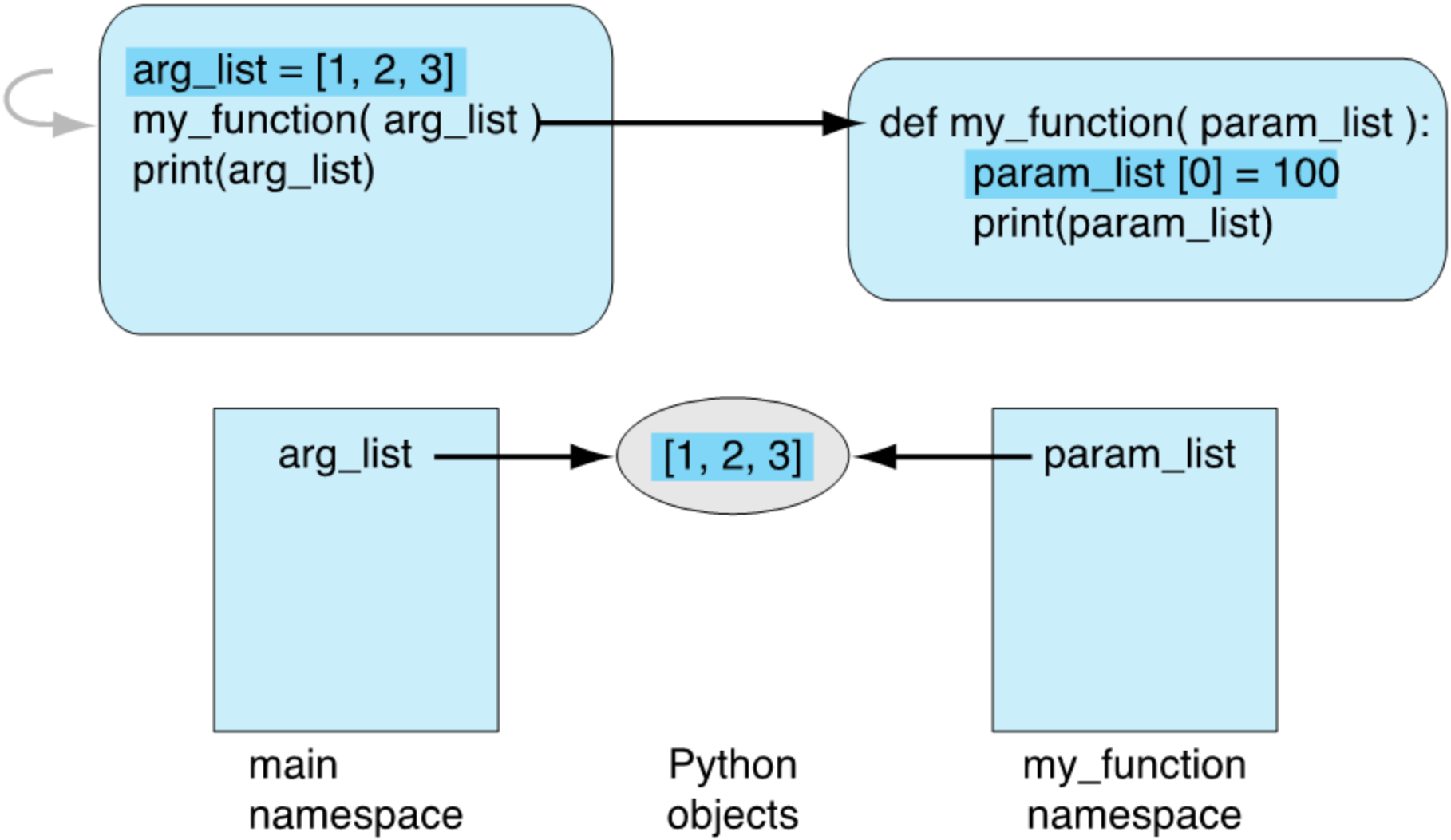

Function namespace: at function start.

# What does “pass” mean?

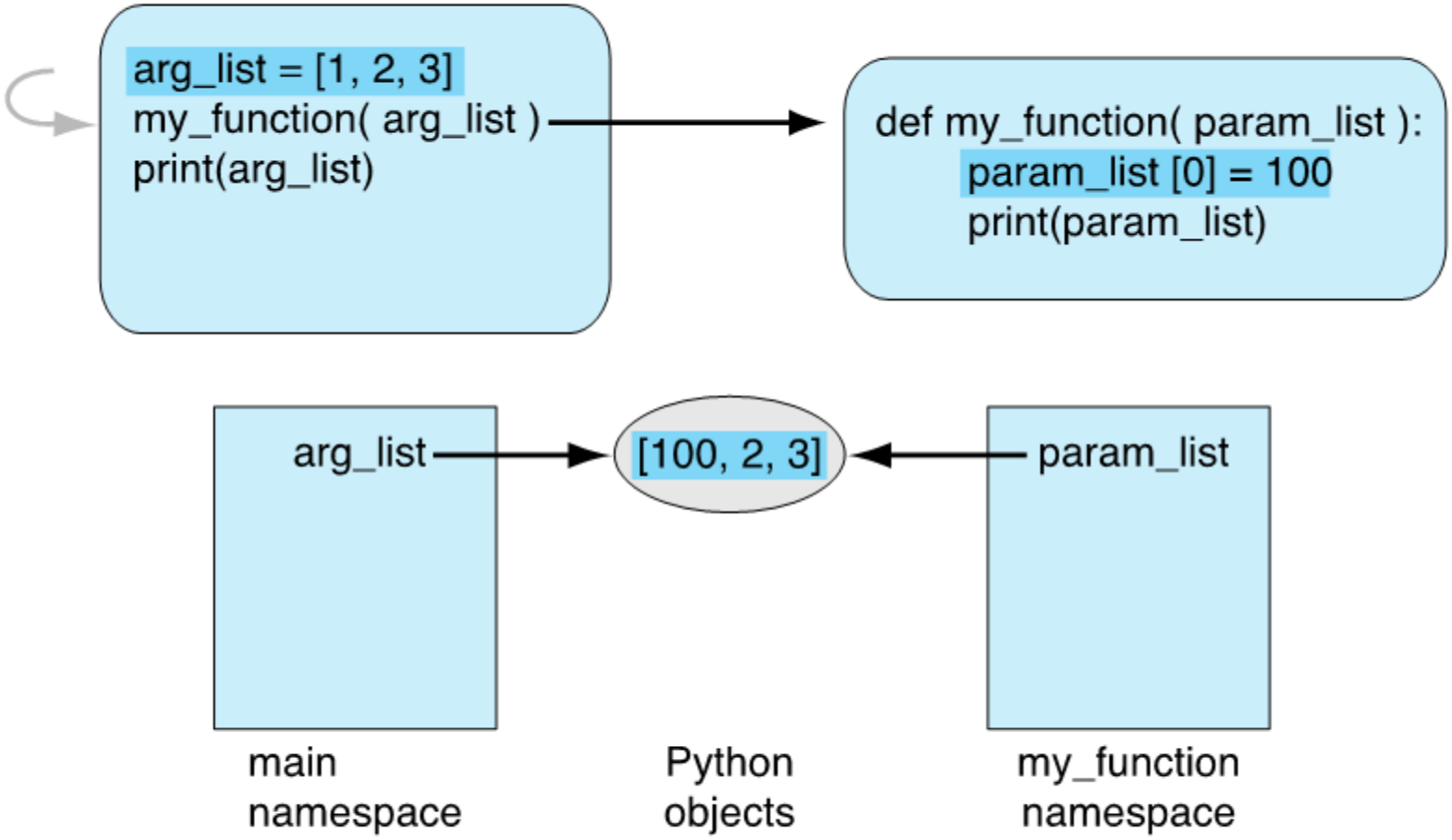
- The diagram should make it clear that the parameter name is local to the function namespace
- Passing means that the argument and the parameter, named in two different namespaces, share an association with the same object
- So “passing” means “sharing” in Python

# passing mutable objects

- When passing mutable data structures, it is possible that if the shared object is directly modified, both the parameter and the argument reflect that change
- Note that the operation must be a mutable change, a change of the object. An assignment is not such a change.




Function namespace with mutable objects: at function start.




Function namespace with mutable objects after `param_list[0]=100`.

# Constructing a Function to Return the Maximum Value in a List

```
def getMax(alist):
    maxSoFar = alist[0]
    for pos in range(1,len(alist)):
        if alist[pos] > maxSoFar:
            maxSoFar = alist[pos]

    return maxSoFar
```

```
def getMax(alist):
    maxSoFar = alist[0]
    for item in alist[1:]:
        if item > maxSoFar:
            maxSoFar = item

    return maxSoFar
```

# Mean

- The most common measure of central tendency is the *mean* also called the average.
- To compute the mean, we simply add up the values in a collection and divide by the number of items.
- Python has built in function, sum, that return the sum of values in the data set.

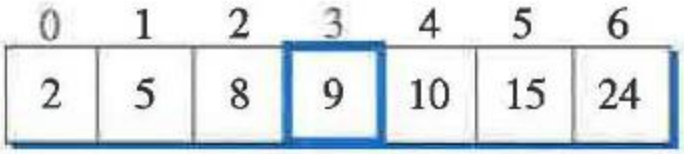
```
def mean(alist):
    mean = sum(alist) / len(alist)
    return mean
```

```
>>> mean([20,32,21,26,33,22,18])
24.571428571428573
>>>
```

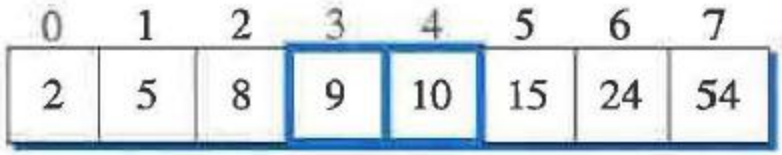
# Median

- *Median*, found by locating the item that occurs in the exact middle of a collection.
- One way to compute the median is to put the items in order, from lowest to highest, and then find the value that falls in the middle.
- If there is an odd number of items, this will be a distinct value.
- If there is an even number of items, the average of the two middle values will be used.

# Median (cont'd.)




The median is 9




The median is 9.5

# Median (cont'd.)

- The process of ordering values is often referred to as *sorting*.
- Sorting is a very important topic in computer science and there are a number of algorithms that can be used to take a list of data and place them in order.

# Median (cont'd.)

```
>>> alist = [24,5,8,2,9,15,10]
>>> alist.sort()
>>> alist
[2, 5, 8, 9, 10, 15, 24]
>>> len(alist)
7
>>> len(alist)//2
3
>>> alist[3]
9
>>> alist = [2,5,8,9,10,15,24,54]
>>> len(alist)
8
>>> len(alist)//2
4
>>> alist[4]
10
>>>
```

# Median (cont'd.)

```
def median(alist):
    copylist = alist[:]  #make a copy using slice operator
    copylist.sort()
    if len(copylist)%2 == 0: #even length
        rightmid = len(copylist)//2
        leftmid = rightmid - 1
        median = (copylist[leftmid] + copylist[rightmid])/2.0
    else:      #odd length
        mid = len(copylist)//2
        median = copylist[mid]
    return median
```

```
>>> median([20,32,21,26,33,22,18])
22
>>>
>>> median([20,32,21,26,33,22,18,29])
24
>>>
```

# Median (cont'd.)

- Before sorting, we are careful to make a copy of the original list in case the original order is important for later processing.

```
def var (anotherNum):
    anotherNum = 5
    print(anotherNum)
```

```
>>> myNum = 3
>>> myNum
3
>>> var(myNum)
5
>>> myNum
3
```

```
def TheList(anotherList):
    anotherList[1] = 3
    print(anotherList)
```

```
>>> myList = [1,1,1]
>>> myList
[1, 1, 1]
>>> TheList(myList)
[1, 3, 1]
>>> myList
[1, 3, 1]
```