The Practice of Computing Using

PYTHON

William Punch  **Richard Enbody**

Chapter 6

Lists and Tuples

Addison-Wesley
is an imprint of
PEARSON

Data Structures

Data Structures and Algorithms

- Part of the “science” in computer science is the design and use of data structures and algorithms.
- As you go on in CS, you will learn more and more about these two areas.

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) 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

The Python List Data Structure

- A list is very simple - it is just an ordered sequence of items.
- You have seen such a sequence before in a string. A string is just a particular kind of list. What kind?

Make a List

- Like all data structures, lists have a constructor, named the same as the data structure. It takes an iterable data structure and adds each item to the list.
- It also has a shortcut: the use of square brackets [ ] to indicate explicit items.

More List Making

```
aList = list('abc')
  aList ['a', 'b', 'c']

newList = [1, 3.14159, 'a', True]
```

Similarities with Strings

- concatenate/+ (but only of lists)
- repeat/*
- indexing (the [ ] operator)
- slicing ([:])
- membership (the in operator)
- len (the length operator)

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 “”.

```
myList = [1, 'a', 3.14159, True]
```

myList

1	'a'	3.14159	True	
0	1	2	3	Index Forward
−4	−3	−2	−1	Index Backward

```
myList[1] → 'a'
myList[:3] → [1, 'a', 3.14159]
```

FIGURE 6.1 The structure of a list.

Indexing

- Can be a little confusing - what does the [ ] mean, a list or an index?

```
[1, 2, 3][1] 2
```

- Context solves the problem. An index always comes at the end of an expression and is preceded by something (a variable, a sequence).

List of Lists

```
myLst = ['a', [1, 2, 3], 'z']
```

- What is the second element (index 1) of that list?
- Another list:

```
myLst[1][0] # apply left to right
[1, 2, 3][0] ⇒ 1
myLst[1] ⇒ [1, 2, 3]
```

Operators

```
[1, 2, 3] + [4] [1, 2, 3, 4]

[1, 2, 3] * 2 [1, 2, 3, 1, 2, 3]

1 in [1, 2, 3] True

[1, 2, 3] < [1, 2, 4] True
```

Compare index to index, the first difference determines the result.

List Functions

- len(lst): Number of elements in list (top level). `len([1, [1, 2], 3])` 3
- min(lst): Minimum element in the list. If list of lists, looks at first element of each list.
- max(lst): Max element in the list.
- sum(lst): Sum the elements, numeric only.

Iteration

```
for element in [1, [1, 2], 'a',True]:
    print element
```

```
1
[1, 2]
'a'
True
```

Mutable

Change an Object's Contents

- Strings are immutable. Once created, the object's contents cannot be changed. New objects can be created to reflect a change, but the object itself cannot be changed:

```
myStr = 'abc'
myStr[0] = 'z'        # cannot do!
# instead, make new str
newStr = myStr.replace('a','z')
```

Lists are Mutable

- Unlike strings, lists are mutable. You can change the object's contents!

```
myLst = [1, 2, 3]
myLst[0] = 127
print myLst [127, 2, 3]
```

List Methods

- Remember, a function is a small program (such as len) that takes some arguments, the stuff in the parenthesis, and returns some value.
- A method is called in a special way, the "dot call". It is called in the context of an object (or a variable holding an object).

Again, Lists have Methods

```
myList = ['a',1,True]
myList.append('z')
```

arguments to the method

the object that we are calling the method with

the name of the method

Some New Methods

- A list is mutable and can change:
  - myList[0]='a' #index assignment
  - myList.append(), myList.extend()
  - myList.pop()
  - myList.insert(), myList.remove()
  - myList.sort()
  - myList.reverse()

More about List Methods

- Most of these methods do not return a value.
- This is because lists are mutable so the methods modify the list directly; there is no need to return anything.
- Can be confusing.

Unusual Results

```
myLst = [4, 7, 1, 2]
myLst = myLst.sort()
myLst == None        # what happened?
```

What happened was the sort operation changed the order of the list in place (right side of assignment). Then the sort method returned None, which was assigned to the variable. The list was lost and None is now the value of the variable.

Range

- We have seen the range function before. It generates a sequence of integers.
- In fact, what it generates is a list with that sequence:

```
myLst = range(1, 5)
myLst [1, 2, 3, 4]
```

Split

- The string method split generates a sequence of characters by splitting the string at certain split-characters.
- It, too, returns a list:

```
splitLst = 'this is a test'.split()
splitLst
```

```
['this', 'is', 'a', 'test']
```

Sorting

- Only lists have a built-in sorting method. Thus you often convert your data to a list if it needs sorting:

```
myLst = list('xyzabc')
myLst → ['x','y','z','a','b','c']
myLst.sort()
# convert back to a string
sortStr = ''.join(myLst)
    → 'abcxyz'
```

Sorted Function

- The sorted function will break a sequence into elements and sort the sequence, placing the results in a list:

```
sortLst = sorted('hi mom')
```

```
[' ','h','i','m','m','o']
```

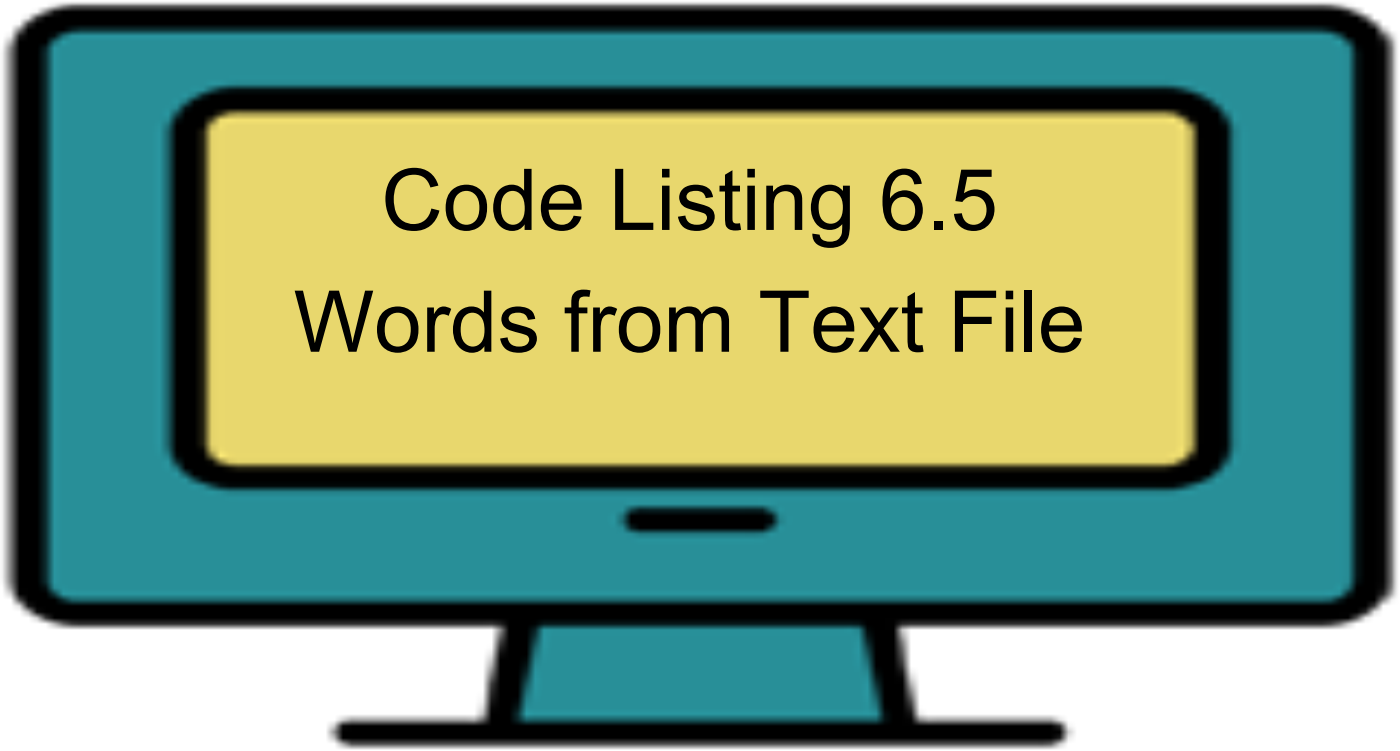
Some Examples

Anagram Example

- Anagrams are words that contain the same letters in a different order. For example: ‘iceman’ and ‘cinema.’
- A strategy to identify anagrams is to take the letters of a word, sort those letters, then compare the sorted sequences.
- Anagrams should have the same sequence.

```
def areAnagrams(word1, word2):
    """Return true, if anagrams
    # Sort the words.
    word1_sorted = sorted(word1)
    word2_sorted = sorted(word2)

    # compare lists.
    if word1_sorted == word2_sorted:
        return True
    else:
        return False
```



```
def makeWordList(gFile):
    """Create a list of words from the file."""
    speech = []                 # list of file words
    for lineString in gFile:
        lineList = lineString.split()
        for word in lineList:
            if word != "--":    # straggler in file
                speech.append(word)
    return speech
```

```
# requires list of words from 6.5
def makeUnique(speech):
    """Create a list of unique words."""
    unique = []  # list of unique words
    for word in speech
        # check first if word is already there
        if word not in unique:
            unique.append(word)
    return unique
```

More about Mutables

Reminder: Assignment

- 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.

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

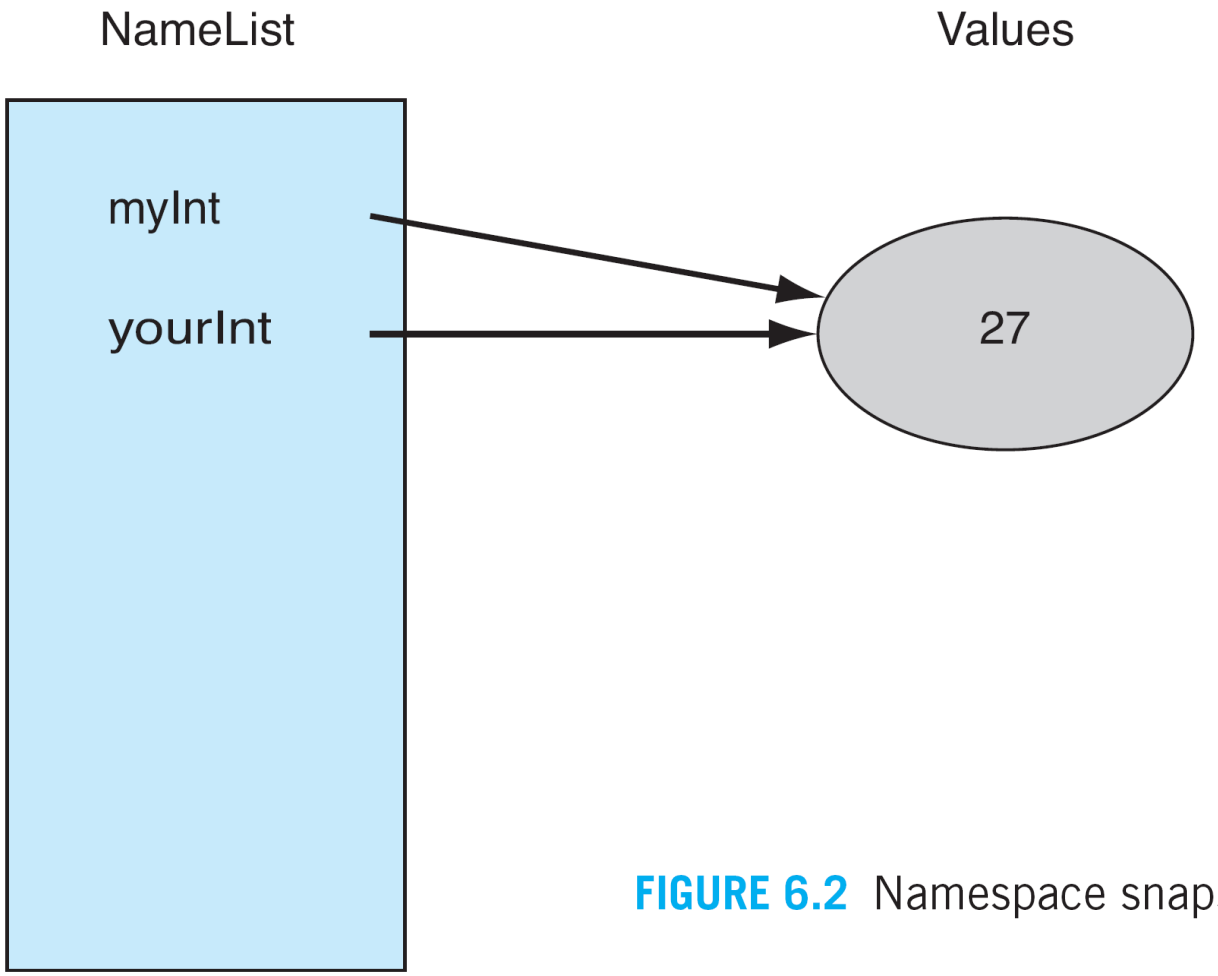


FIGURE 6.2 Namespace snapshot #1.

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
```

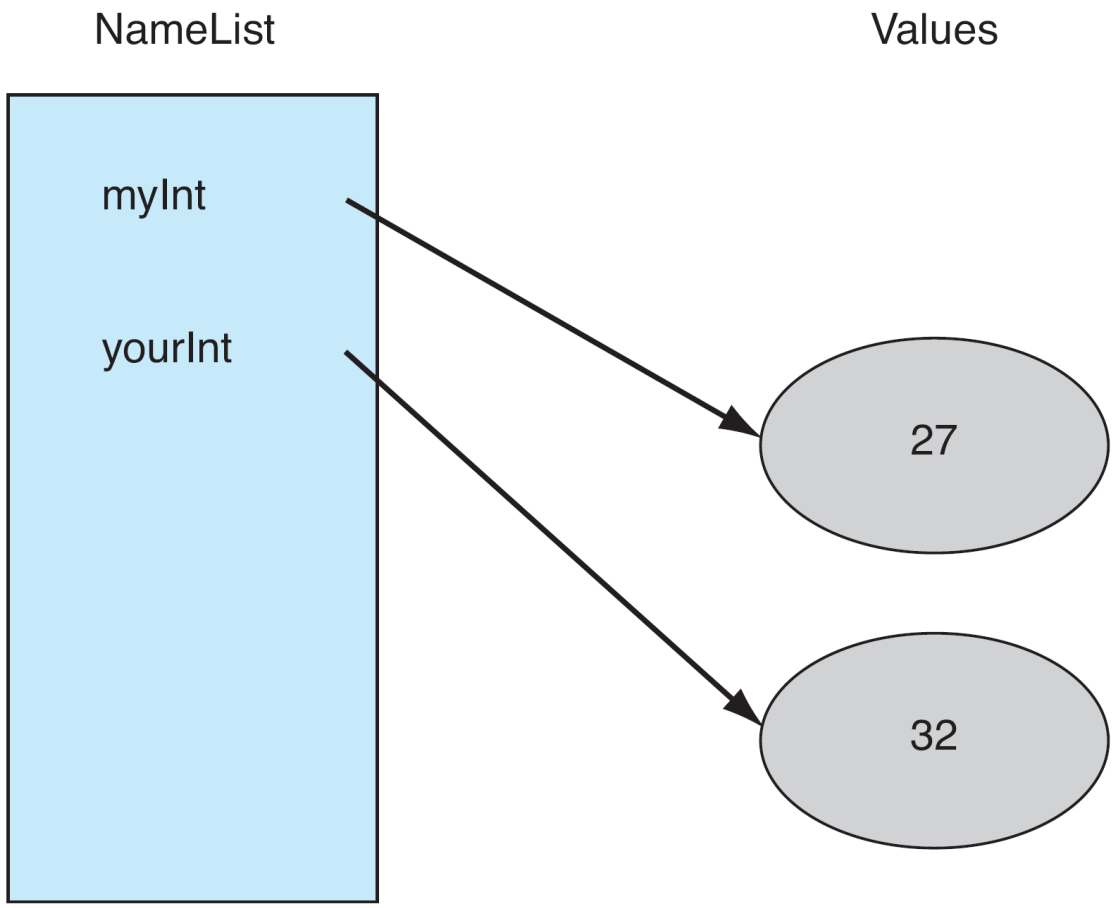


FIGURE 6.3 Modification of a reference to an immutable object.

Mutability Changes an Object

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

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

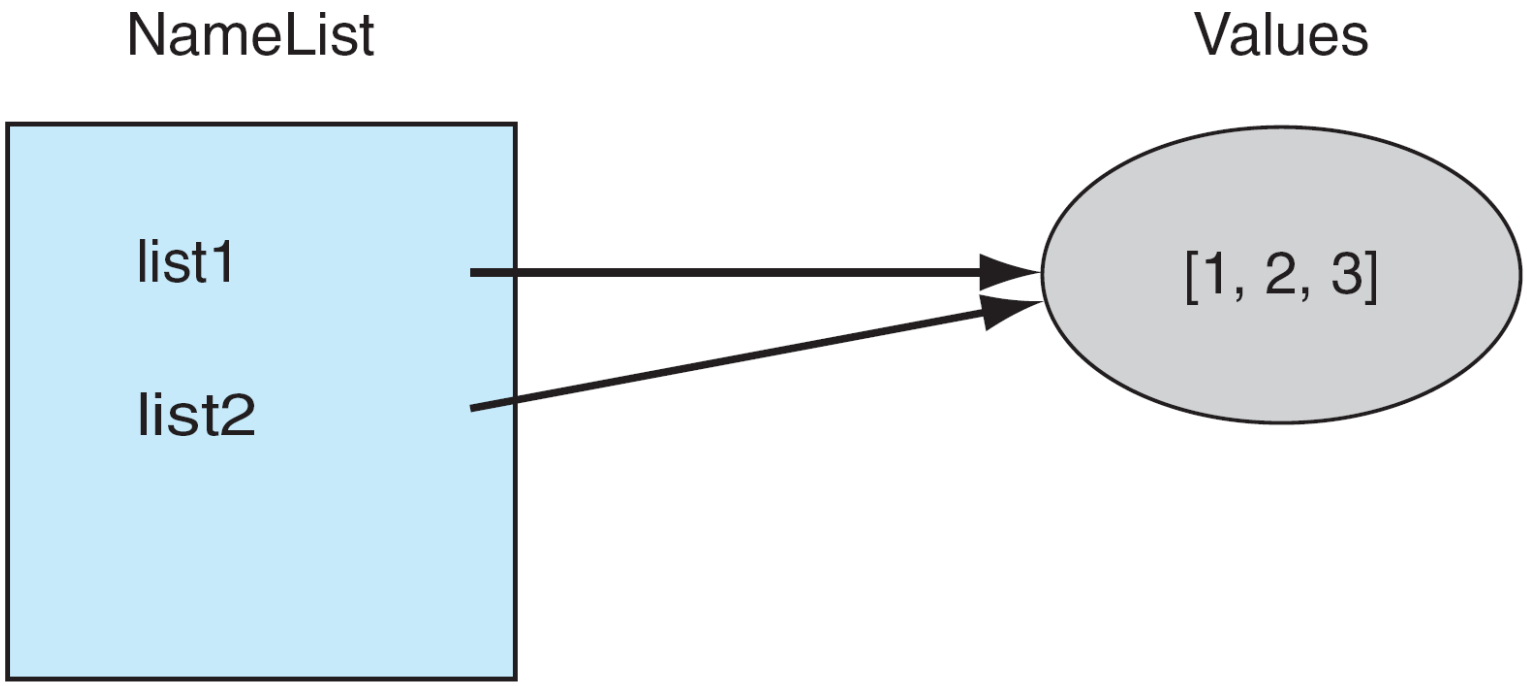


FIGURE 6.4 Namespace snapshot after assigning mutable objects.

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

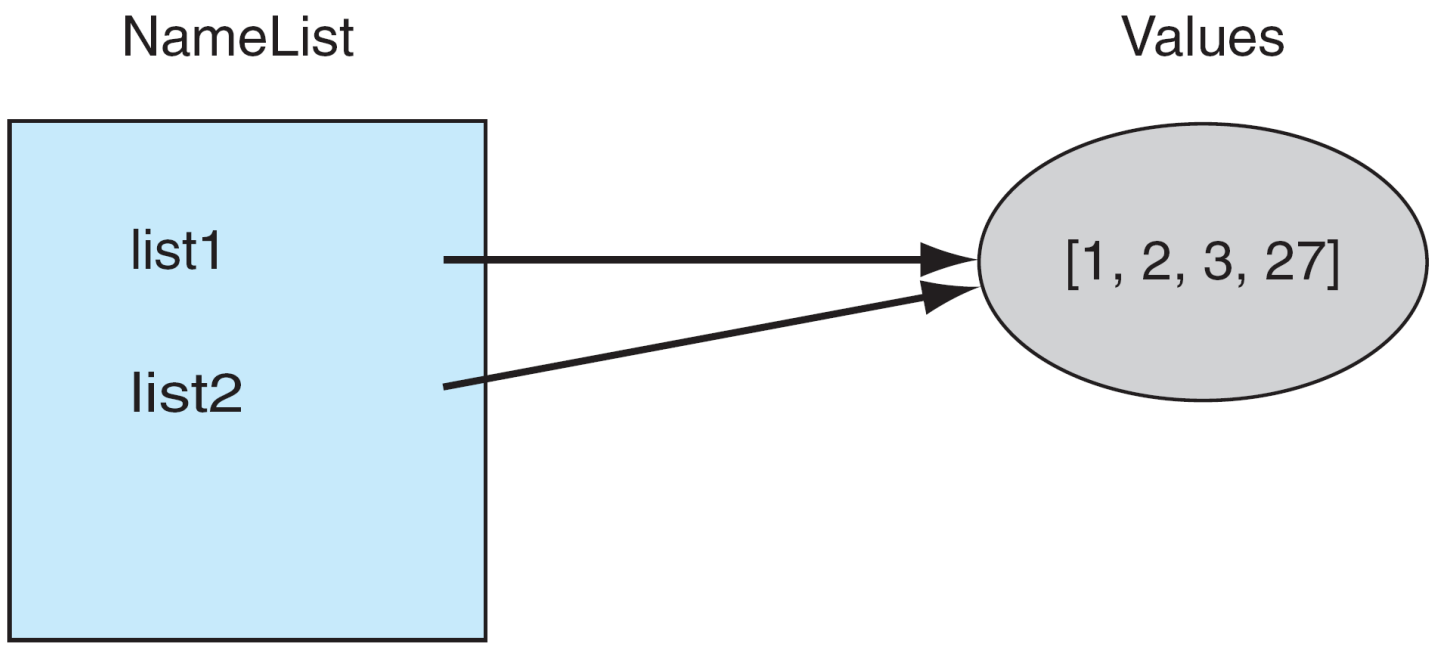


FIGURE 6.5 Modification of shared, mutable objects.

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)
```

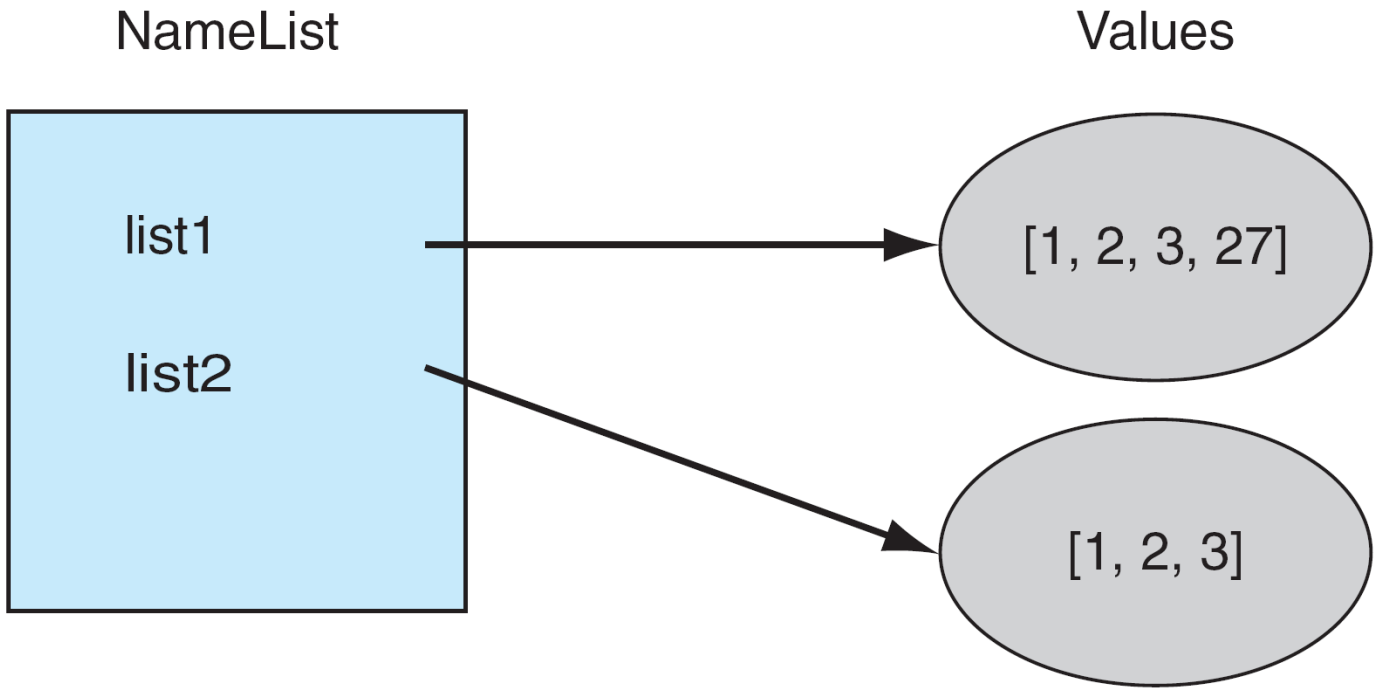


FIGURE 6.6 Making a distinct copy of a mutable object.

The Problem 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**.

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

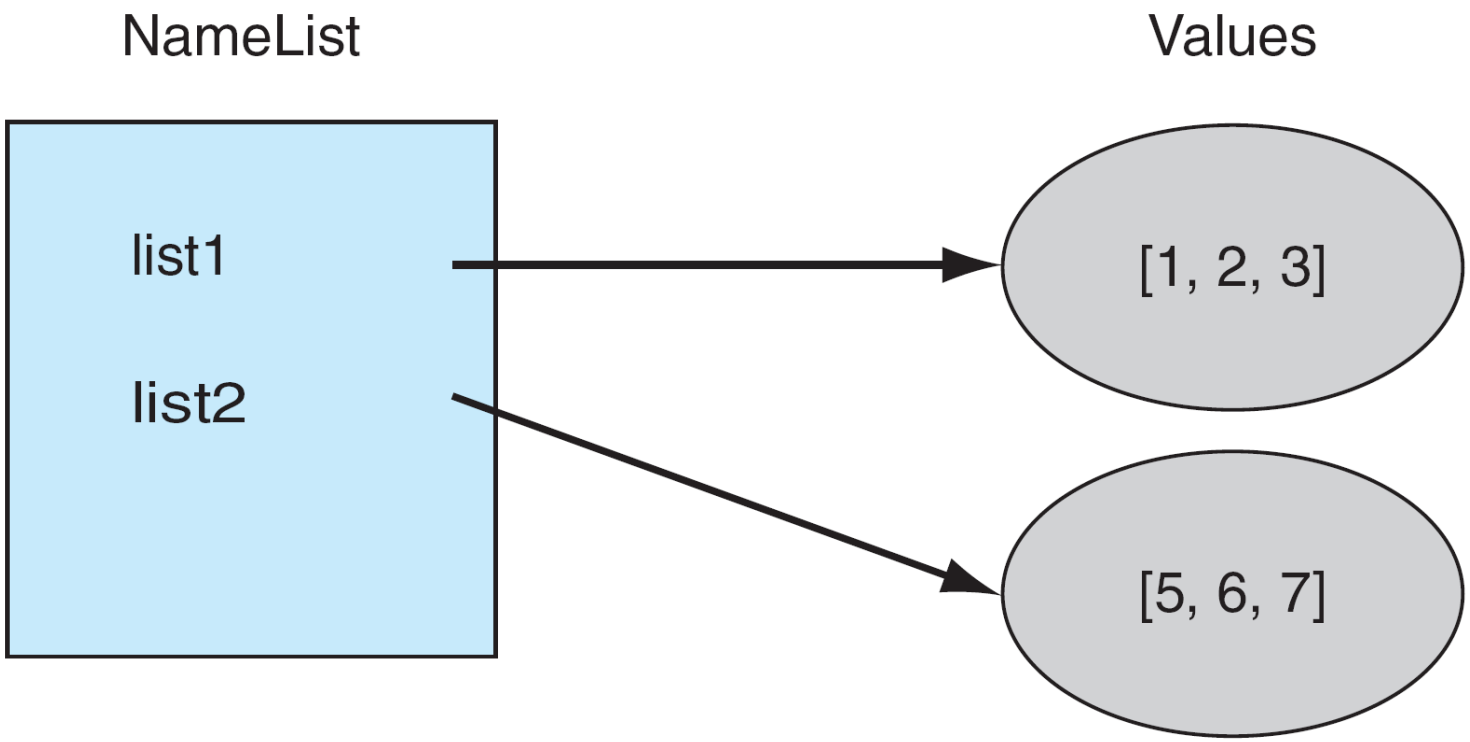


FIGURE 6.8 Simple lists before append.

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

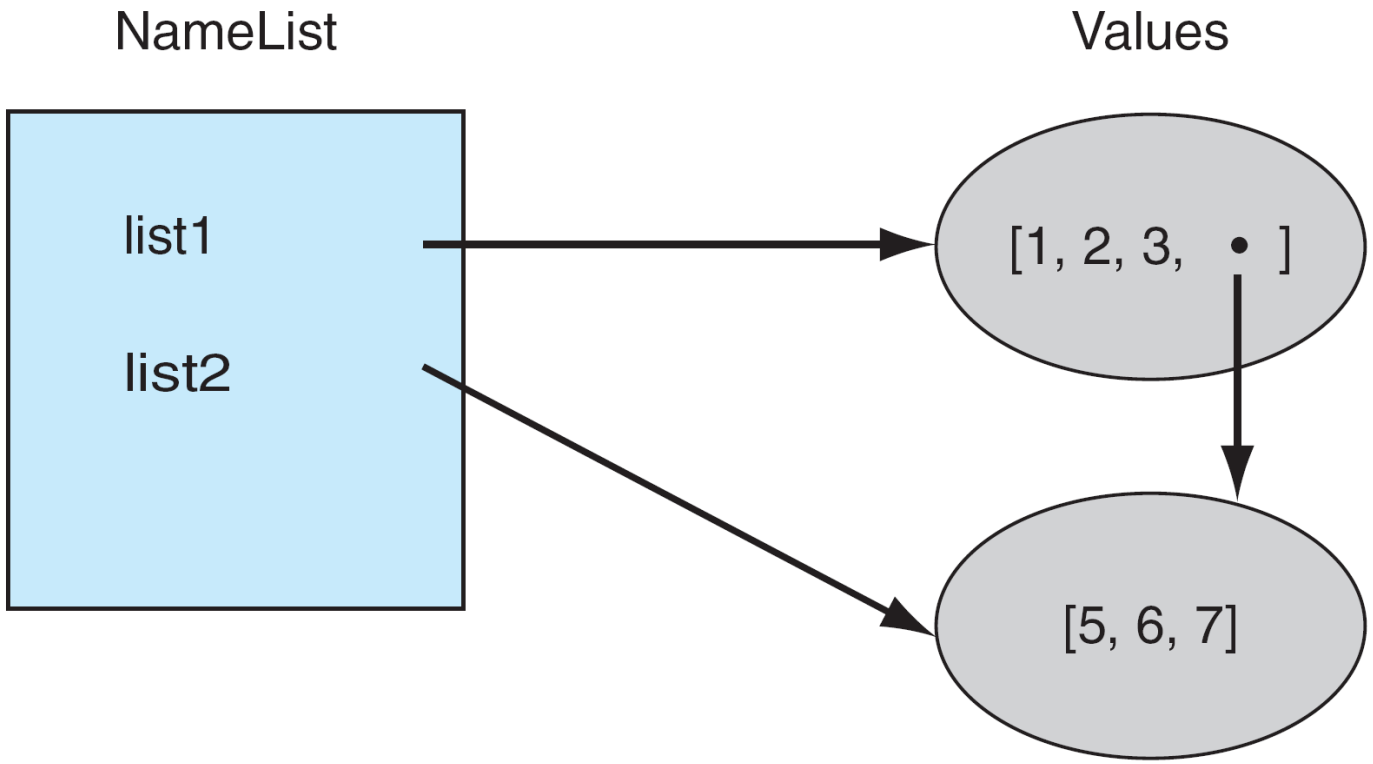


FIGURE 6.9 Lists after append.

```
list1 = [1,2,3]
list2 = [5,6,7]
list1.append(list2)
list3 = list2
list3[2] = 88
```

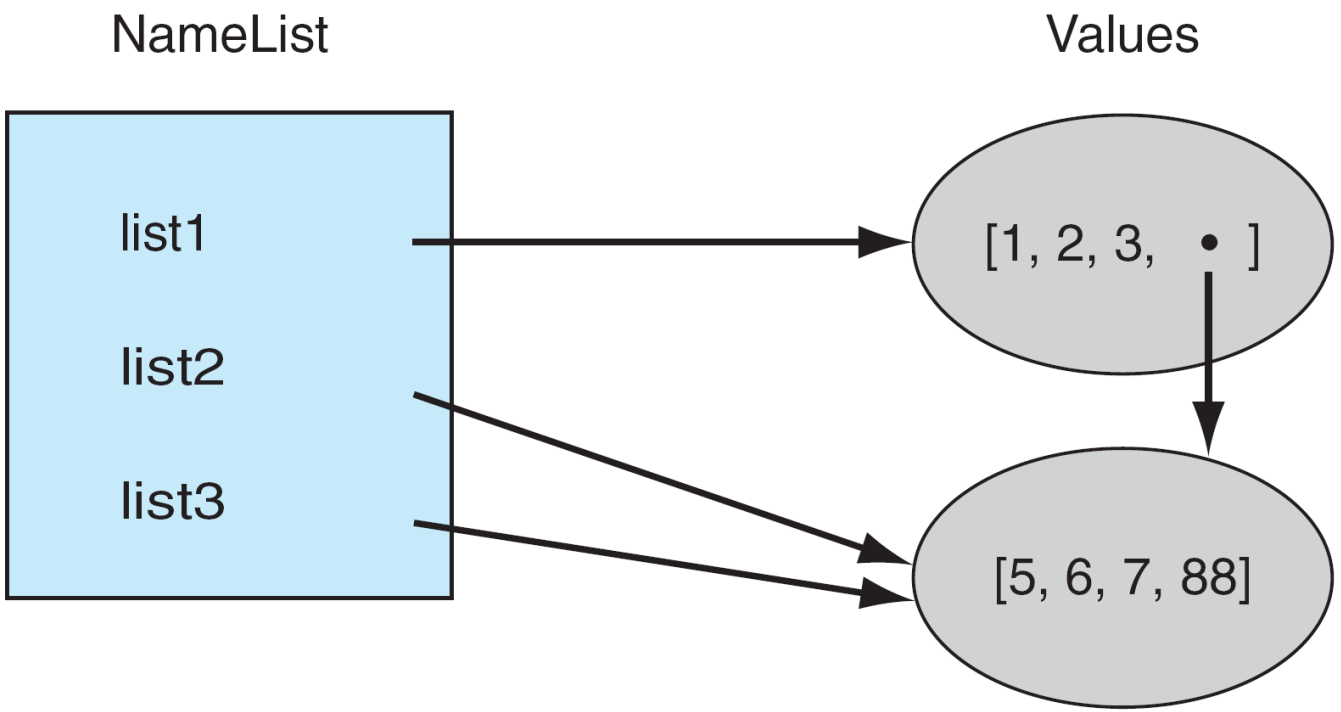


FIGURE 6.10 Final state of copying example.

Tuples

Tuples

- Tuples are easy: they are simply immutable lists.
- They are designated with (,):

```
myTuple = (1,'a',3.14,True)
```

The Question is, Why?

- The real question is, why have an immutable list, a tuple, as a separate type?
- An immutable list gives you a data structure with some integrity, some permanency, if you will.
- You know you cannot accidentally change one.

Lists and Tuples

- Everything that works with a list works with a tuple **except** methods that modify the tuple.
- Thus indexing, slicing, len, print all work as expected.
- However, none of the mutable methods work: append, extend, del.

Commas Make a Tuple

- For tuples, you can think of a comma as the operator that makes a tuple, where the ( ) simply acts as a grouping:

```
myTuple = 1,2  # creates (1,2)
myTuple = (1,) # creates (1)
myTuple = (1)  # creates 1 not (1)
myTuple = 1,     # creates (1)
```

Data Structures In General

Organization of Data

- We have seen strings, lists and tuples so far.
- Each is an organization of data that is useful for some things, not as useful for others.

A Good Data Structure

- Efficient with respect to us (some algorithms).
- Efficient with respect to the amount of space used.
- Efficient with respect to the time it takes to perform some operations.

EPA Example

List Comprehensions

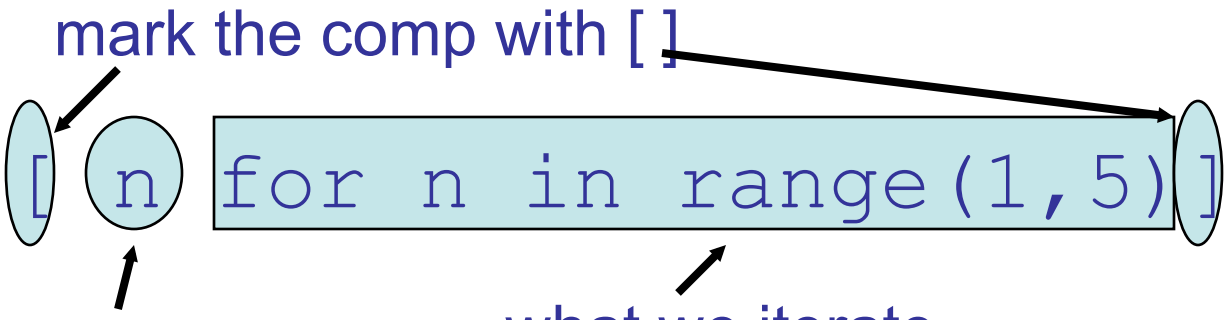
Lists are a Big Deal!

- The use of lists in Python is a major part of its power.
- Lists are very useful and can be used to accomplish many tasks.
- Therefore Python provides some pretty powerful support to make common list tasks easier.

Constructing Lists

- One way is a “list comprehension”

```
[n for n in range(1,5)]
```



returns [1,2,3,4]

Modifying What We Collect

```
[ n**2 for n in range(1,6)]
```

- Returns [1,4,9,16,25]. Note that we can only change the values we are iterating over, in this case n.

Multiple Collects

```
[x+y for x in range(1,4) for y in range
  (1,4)]
```

It is as if we had done the following:

```
myList = [ ]
for x in range (1,4):
  for y in range (1,4):
      myList.append(x+y)
```

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

Modifying What Gets Collected

```
[c for c in “Hi There Mom” if c.isupper()]
```

- The “if” part of the comprehensive controls which of the iterated values is collected at the end. Only those values which make the if part true will be collected:

  [‘H’,’T’,’M’]