INT3075 Programming and Problem Solving for Mathematics

Lists and Tuples

Data Structures

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

The Python List Data Structure

- a list is 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.

make a list

```
>>> a_list = [1,2,'a',3.14159]
>>> week_days_list = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
>>> list_of_lists = [ [1,2,3], ['a','b','c']]
>>> list_from_collection = list('Hello')
>>> a_list
[1, 2, 'a', 3.1415899999999999]
>>> week_days_list
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
>>> list_of_lists
[[1, 2, 3], ['a', 'b', 'c']]
>>> list_from_collection
['H', 'e', 'l', 'l', 'o']
>>> []
[]
>>>
```

Similarities with strings

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

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
```

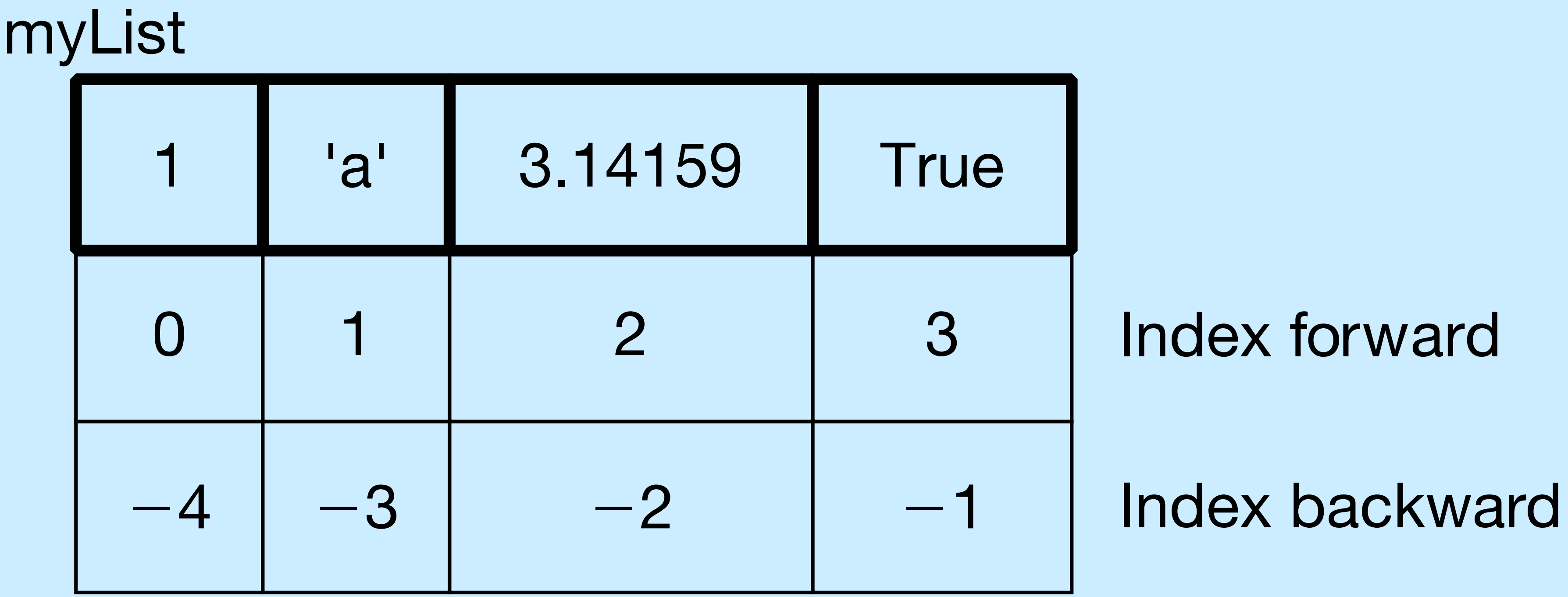
```
[1, 2, 3] < [1, 2, 4] ⇒ True
```

compare index to index, first difference determines the result

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 ' '

```
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 7.1 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. Index always comes at the end of an expression, and is preceded by something (a variable, a sequence)

List of Lists

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

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

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

List Functions

- `len(lst)`: number of elements in list (top level). `len([1, [1, 2], 3])` ⇒ `3`
- `min(lst)`: smallest element. Must all be the same type!
- `max(lst)`: largest element, again all must be the same type
- `sum(lst)`: sum of the elements, numeric only

Iteration

You can iterate through the elements of a list like you did with a string:

```
>>> my_list = [1,3,4,8]
>>> for element in my_list:     # iterate through list elements
        print(element ,end=' ') # prints on one line

1 3 4 8
>>>
```

Mutable

Change an object's content

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

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

Lists are mutable

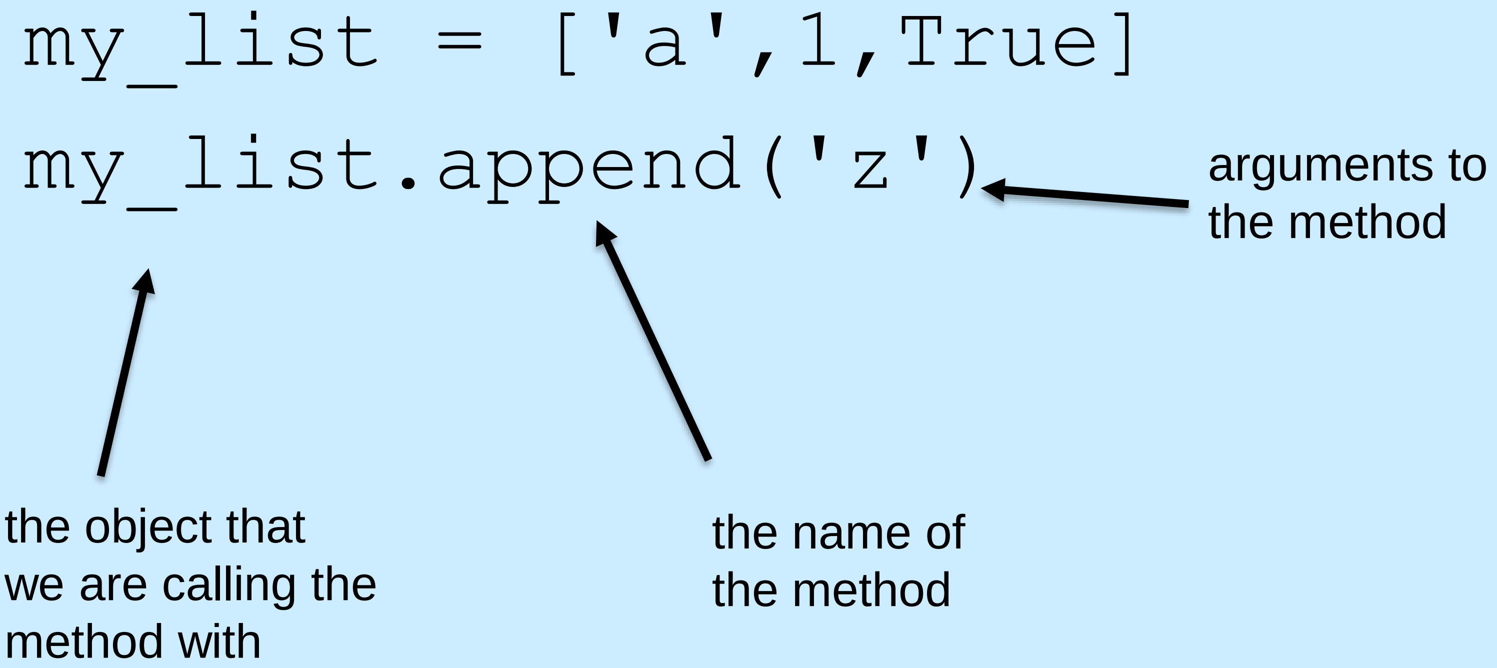
Unlike strings, lists are mutable. You ***can*** change the object's contents!

```
my_list = [1, 2, 3]
my_list[0] = 127
print(my_list) ⇒ [127, 2, 3]
```

List methods

- Remember, a function is a small program that takes some arguments, the stuff in the parenthesis, and returns some value
- a method is a function called in a special way, the ***dot call***. It is called in the context of an object (or a variable associated with an object)

Again, lists have methods



```
my_list = ['a',1,True]
my_list.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:
  - `my_list[0]='a'  #index assignment`
  - `my_list.append(), my_list.extend()`
  - `my_list.pop()`
  - `my_list.insert(), my_list.remove()`
  - `my_list.sort()`
  - `my_list.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. No need to return anything

Unusual results

```
my_list = [4, 7, 1, 2]
my_list = my_list.sort()
my_list ⇒ 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:

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

Split

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

```
split_list = 'this is a test'.split()
split_list
     ⇒ ['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

```
my_list = list('xyzabc')
my_list →['x','y','z','a','b','c']
my_list.sort()   # no return
my_list →
    ['a', 'b', 'c', 'x', 'y', 'z']
```

reverse words in a string

join method of string places the calling string between every element of a list

```
>>> my_str = 'This is a test'
>>> string_elements = my_str.split()            # list of words
>>> string_elements
['This', 'is', 'a', 'test']
>>> reversed_elements = []
>>> for element in string_elements:             # for each word
...     reversed_elements.append(element[::-1]) # reverse , append
...
>>> reversed_elements
['sihT', 'si', 'a', 'tset']
>>> new_str = ' '.join(reversed_elements)  # join with space separator
>>> new_str
'sihT si a tset'                           # each words reversed
>>>
```

Sorted function

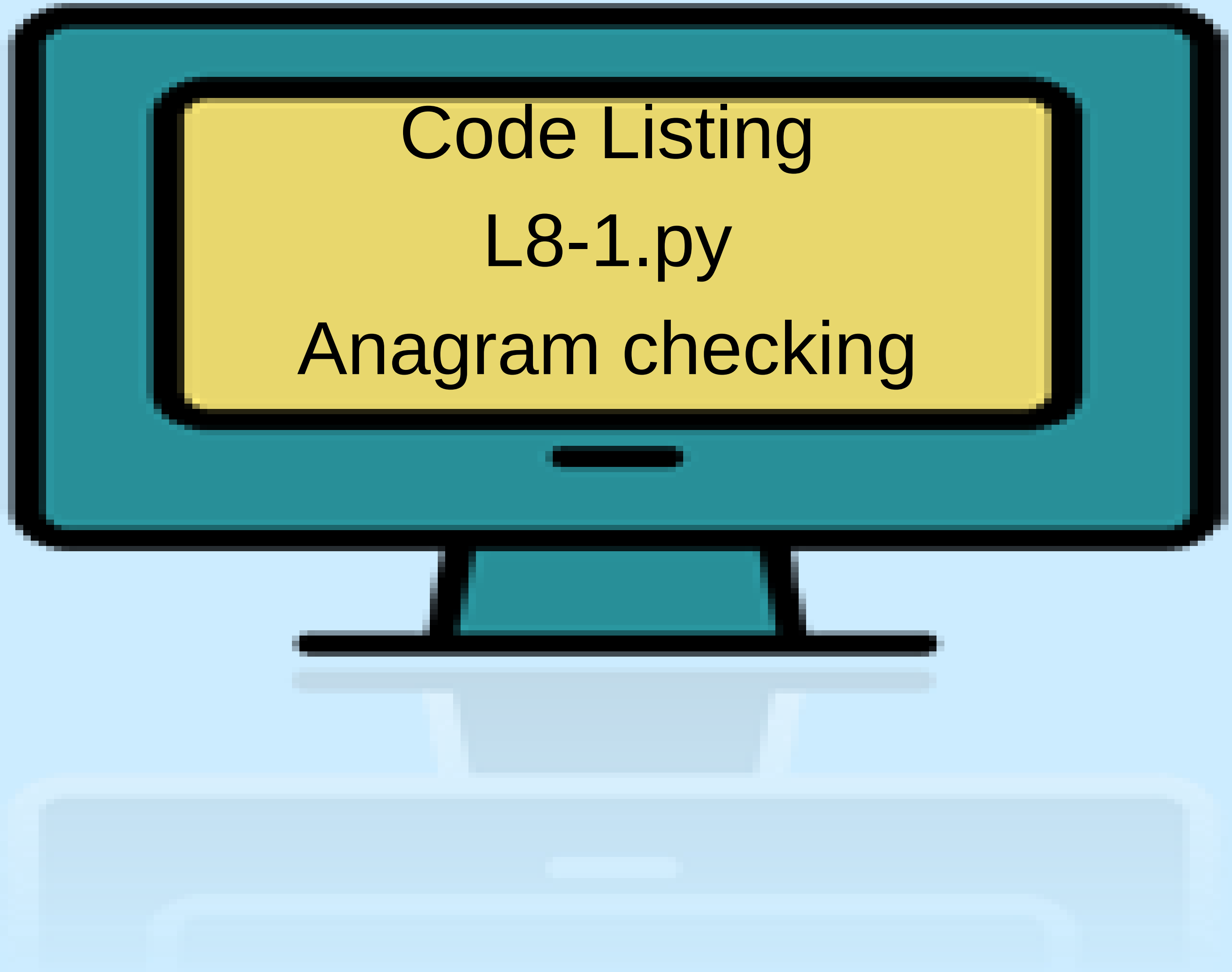
The `sorted` function will break a sequence into elements and sort the sequence, placing the results in a list

```
sort_list = sorted('hi mom')
sort_list ⇒
  [' ','h','i','m','m','o']
```

Some Examples

Anagram example

- Anagrams are words that contain the same letters arranged in a different order. For example: 'iceman' and 'cinema'
- 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 sorted sequence



Code Listing

L8-1.py

Anagram checking

```python
def are_anagrams(word1, word2):
    """Return True, if words are anagrams."""
    #2. Sort the characters in the words
    word1_sorted = sorted(word1)    # sorted returns a sorted list
    word2_sorted = sorted(word2)

    #3. Check that the sorted words are identical.
    if word1_sorted == word2_sorted:  # compare sorted lists
        return True
    else:
        return False
```

Code Listing

L8-2.py

Full Program

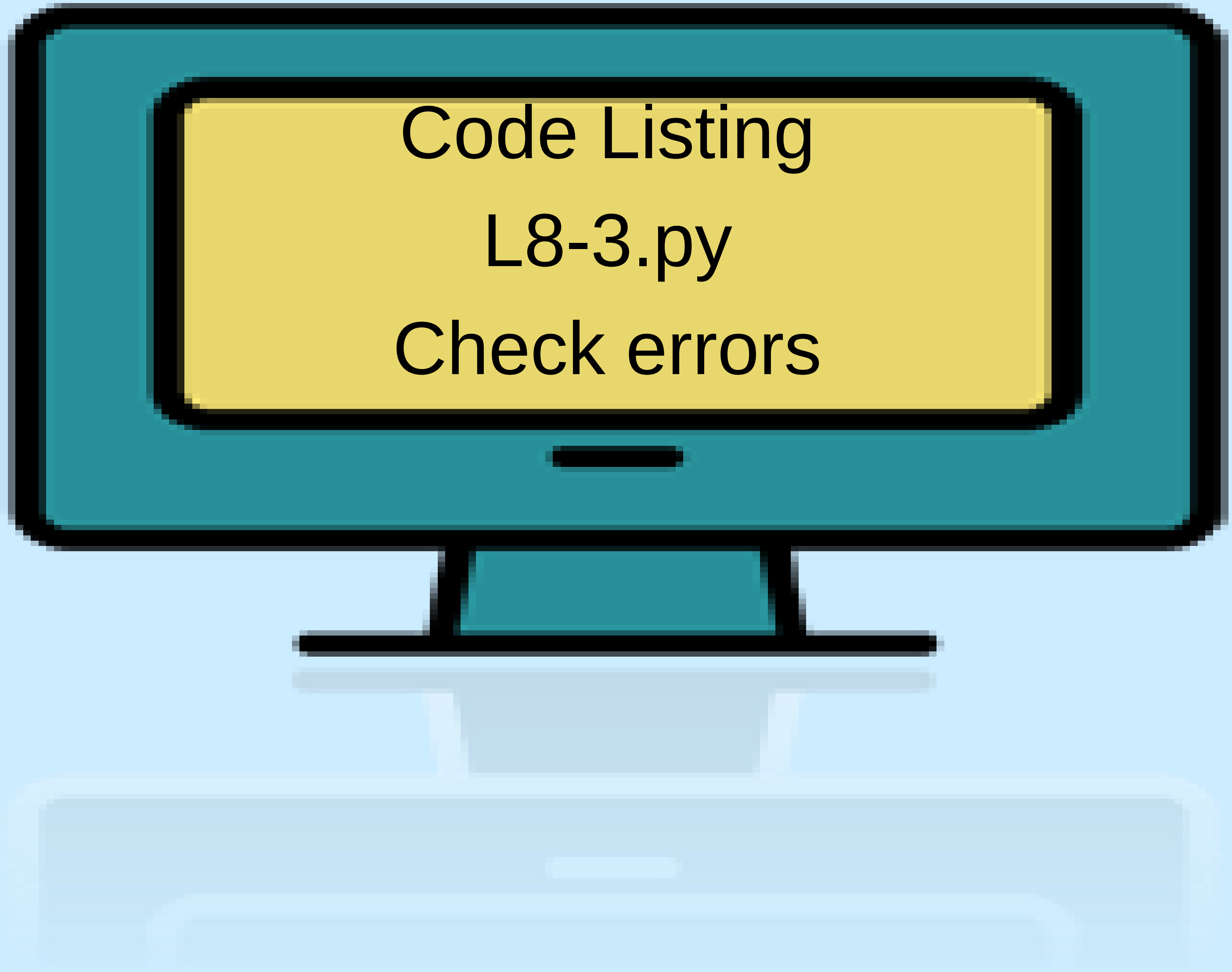
```
def are_anagrams(word1, word2):
    """Return True, if words are anagrams."""
    #2. Sort the characters of the words.
    word1_sorted = sorted(word1)    # sorted returns a sorted list
    word2_sorted = sorted(word2)

    #3. Check that the sorted words are identical.
    return word1_sorted == word2_sorted

print("Anagram Test")

# 1. Input two words.
two_words = input("Enter two space separated words: ")
word1,word2 = two_words.split()  # split into a list of words

if are_anagrams(word1, word2):   # return True or False
    print("The words are anagrams.")
else:
    print("The words are not anagrams.")
```



Code Listing

L8-3.py

Check errors

repeat input prompt for valid input

```
valid_input_bool = False
while not valid_input_bool:
    try:
        two_words = input("Enter two …")
        word1, word2 = two_words.split()
        valid_input_bool = True
    except ValueError:
        print("Bad Input")
```

only runs when no error, otherwise go around again

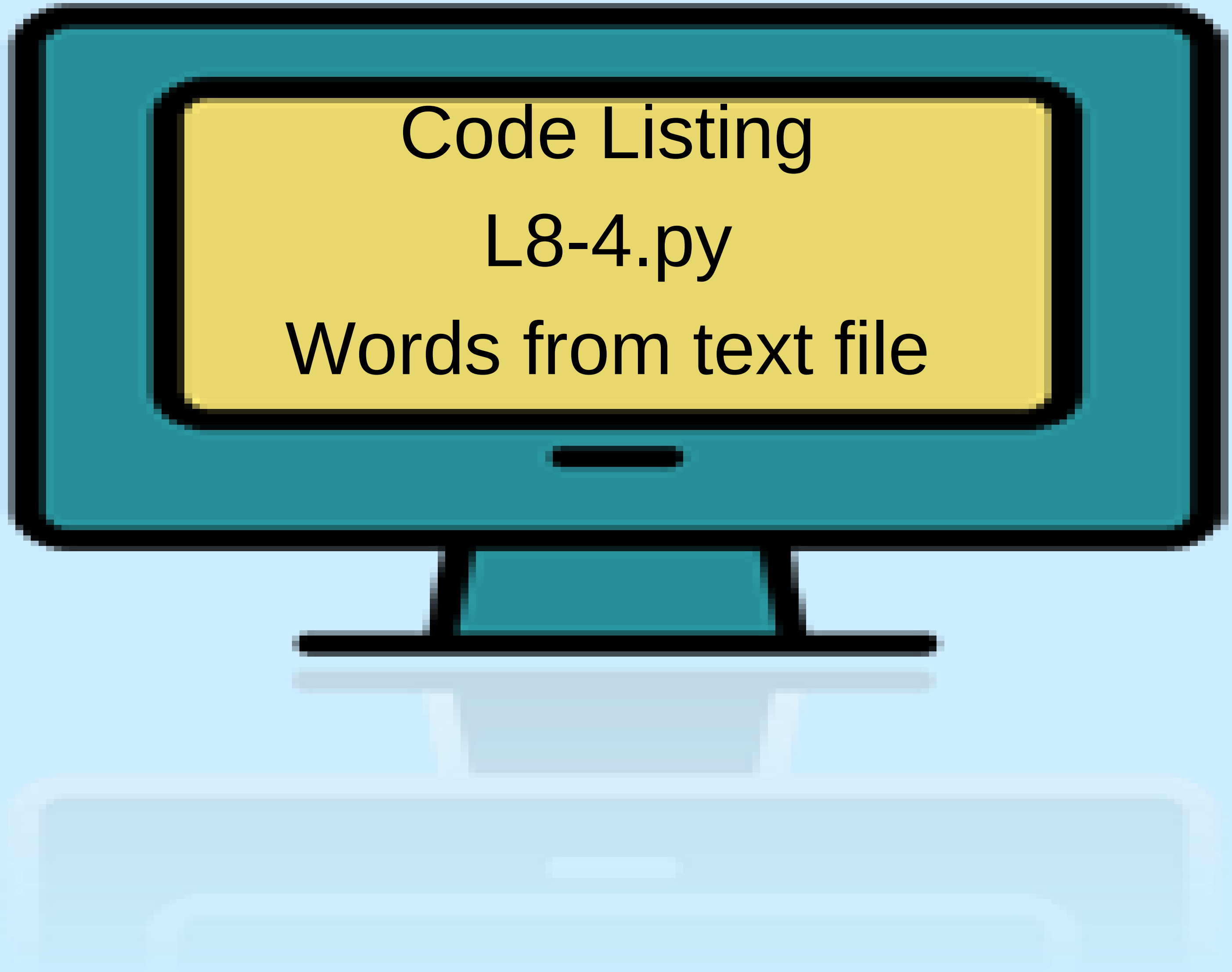
```
def are_anagrams(word1, word2):
    """Return True, if words are anagrams."""
    #2. Sort the characters of the words.
    word1_sorted = sorted(word1)     # sorted returns a sorted list
    word2_sorted = sorted(word2)

    #3. Check that the sorted words are identical.
    return word1_sorted == word2_sorted

print("Anagram Test")

# 1. Input two words, checking for errors now
valid_input_bool = False
while not valid_input_bool:
    try:
        two_words = input("Enter two space separated words: ")
        word1,word2 = two_words.split()   # split the input string into a list
                                            of words
        valid_input_bool = True
    except ValueError:
        print("Bad Input")

if are_anagrams(word1, word2):   # function returned True or False
    print("The words {} and {} are anagrams.".format(word1, word2))
else:
    print("The words {} and {} are not anagrams.".format(word1, word2))
```



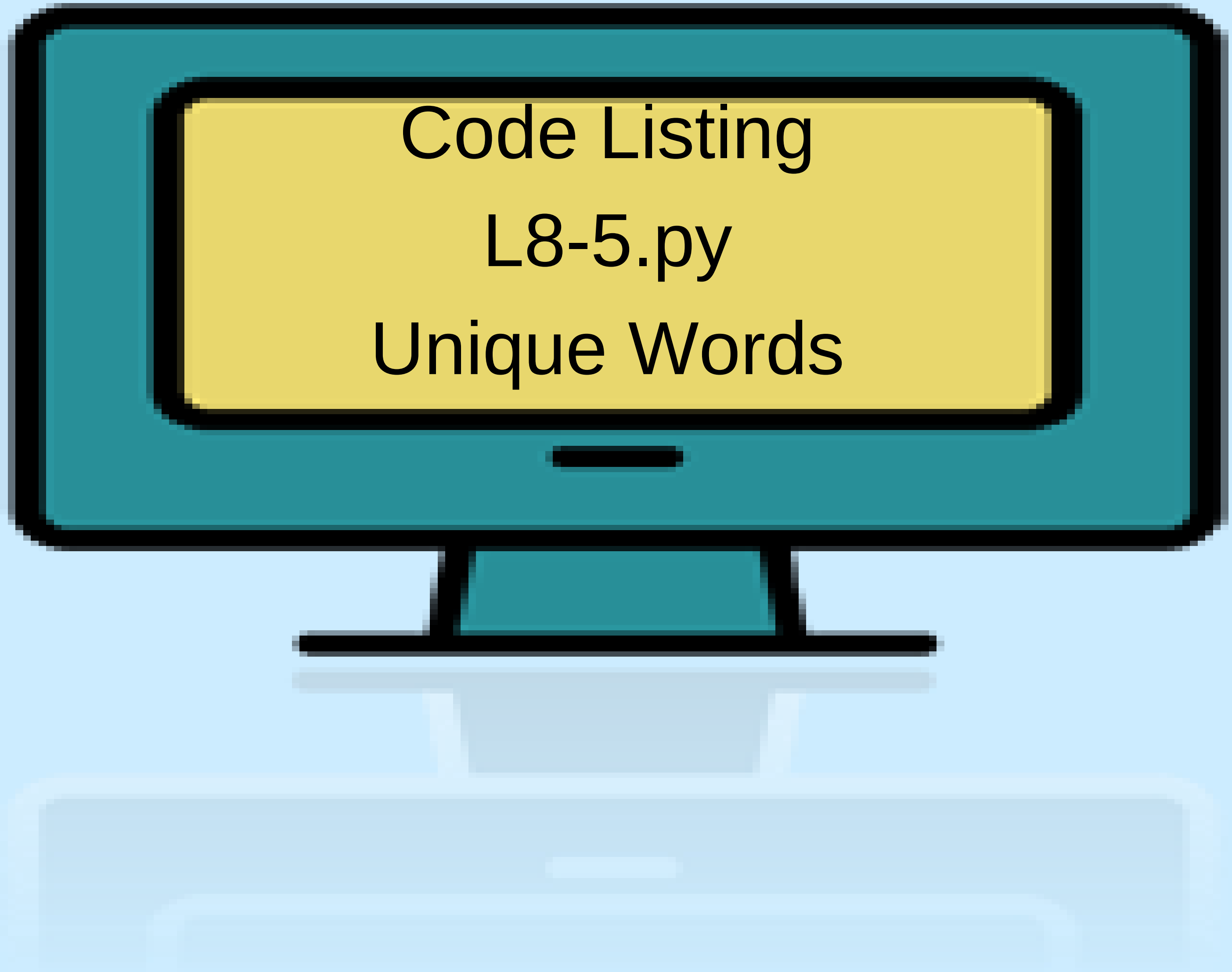
Code Listing

L8-4.py

Words from text file

```
def make_word_list(a_file):
    """Create a list of words from the file."""
    word_list = []      # list of speech words: initialized to be empty

    for line_str in a_file:           # read file line by line
        line_list = line_str.split()  # split each line into a list of words
        for word in line_list:        # get words one at a time from list
            if word != "--":          # if the word is not "——"
                word_list.append(word)   # add the word to the speech list
    return word_list
```



Code Listing

L8-5.py

Unique Words

```python
# Gettysburg address analysis
# count words, unique words

def make_word_list(a_file):
    """Create a list of words from the file."""
    word_list = []      # list of speech words: initialized to be empty

    for line_str in a_file:             # read file line by line
        line_list = line_str.split()    # split each line into a list of words
        for word in line_list:          # get words one at a time from list
            if word != "--":            # if the word is not "——"
                word_list.append(word)    # add the word to the speech list
    return word_list

def make_unique(word_list):
    """Create a list of unique words."""
    unique_list = []  # list of unique words: initialized to be empty

    for word in word_list:          # get words one at a time from speech
        if word not in unique_list: # if word is not already in unique list,
            unique_list.append(word)# add word to unique list

    return unique_list

################################

gba_file = open("gettysburg.txt", "r")
speech_list = make_word_list(gba_file)

# print the speech and its lengths
print(speech_list)
print("Speech Length: ", len(speech_list))
print("Unique Length: ", len(make_unique(speech_list)))
```

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 LHS
- When you assign one variable to another, you ***share the association*** with the same object

```
my_int = 27
your_int = my_int
```

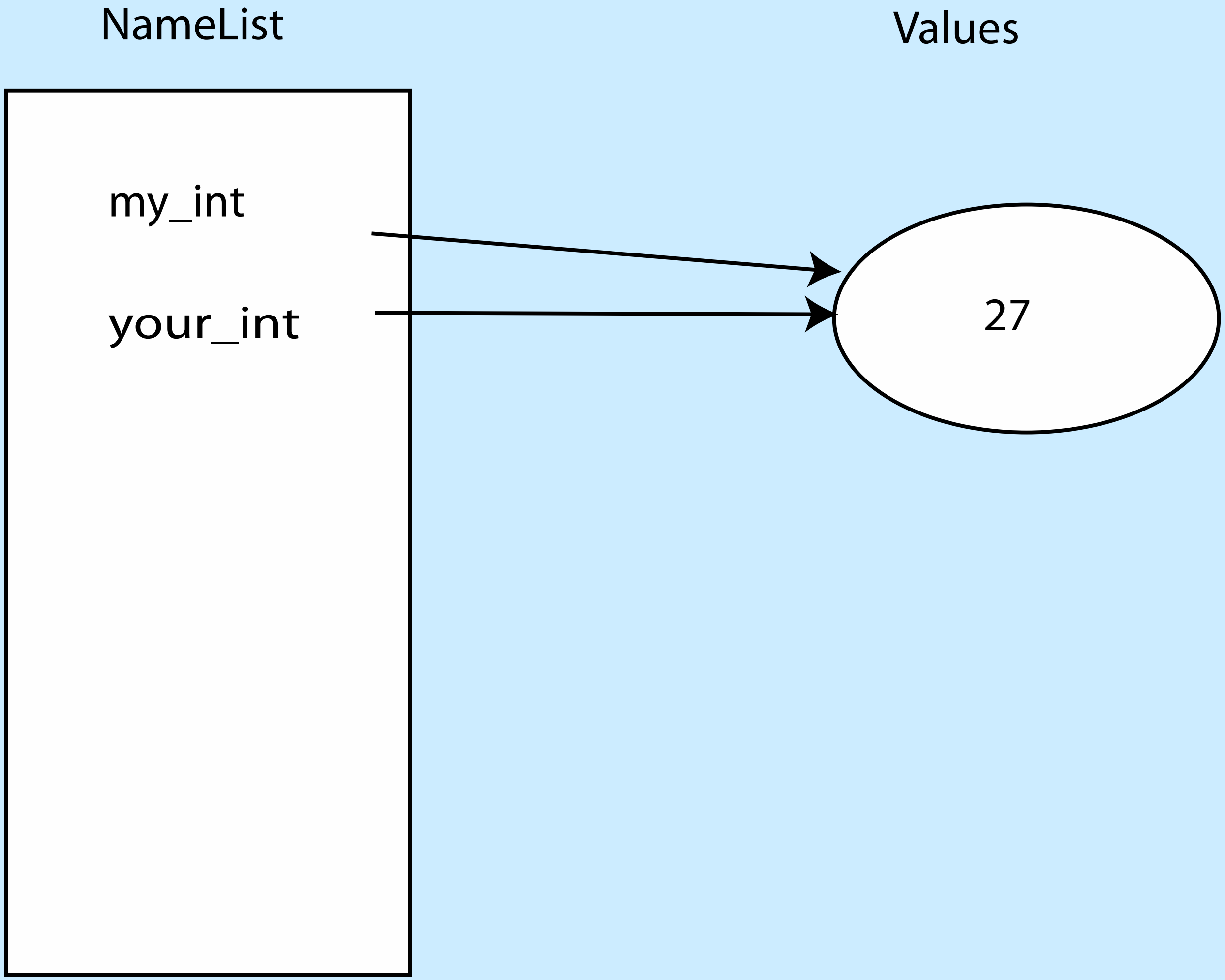
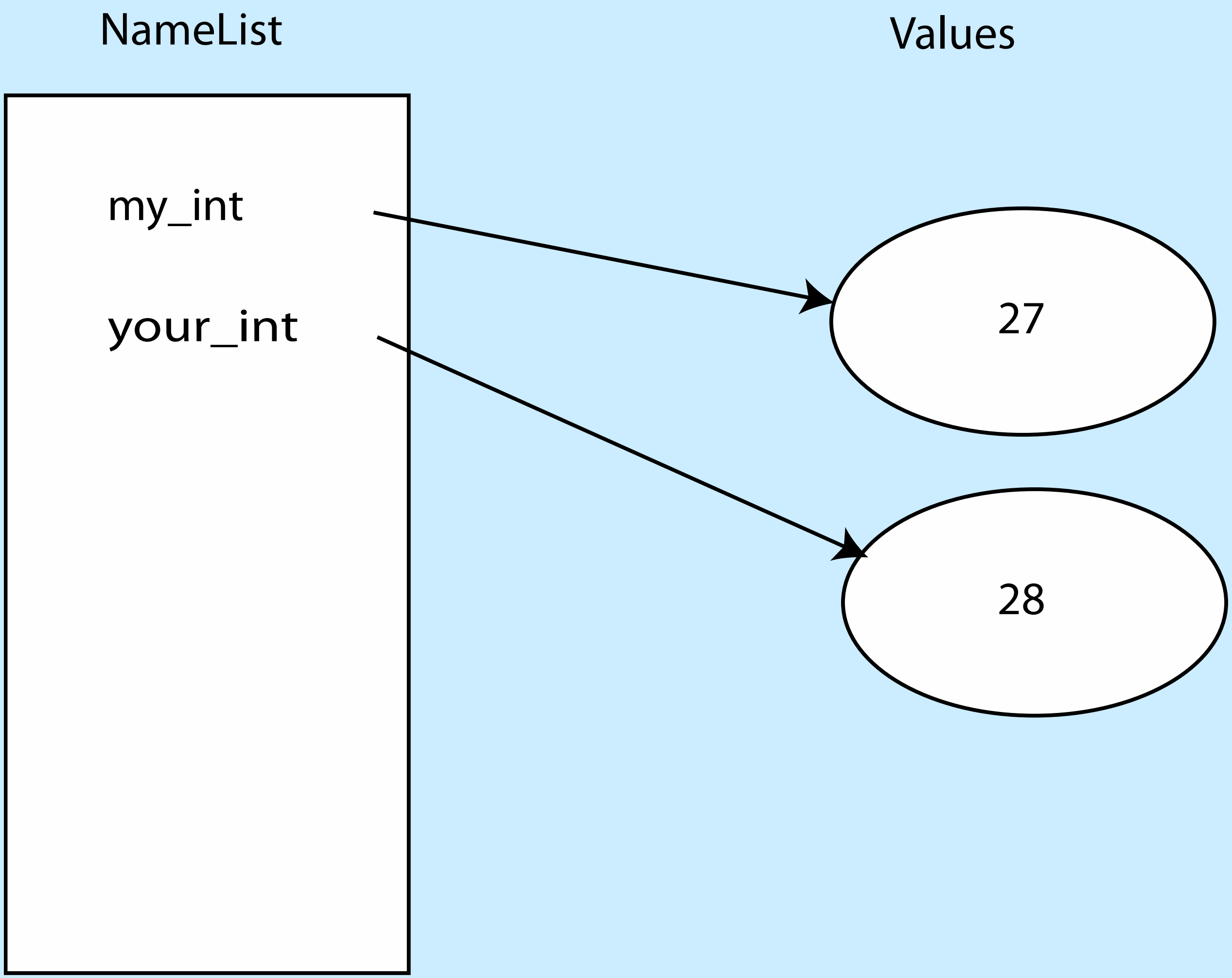


FIGURE 7.2 Namespace snapshot #1.

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.

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



NameList

Values

my_int

your_int

27

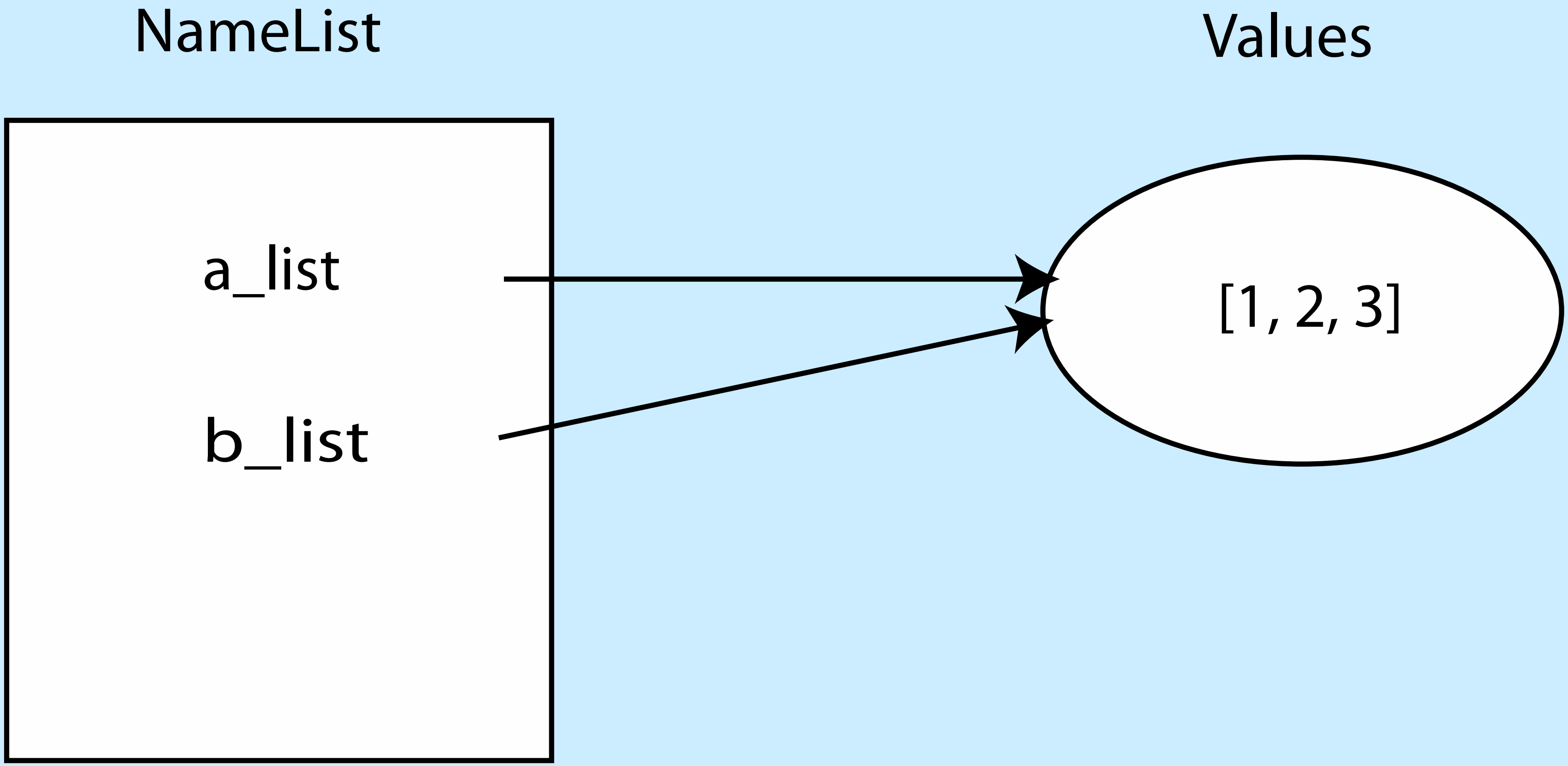
28

FIGURE 7.3 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
```



NameList

Values

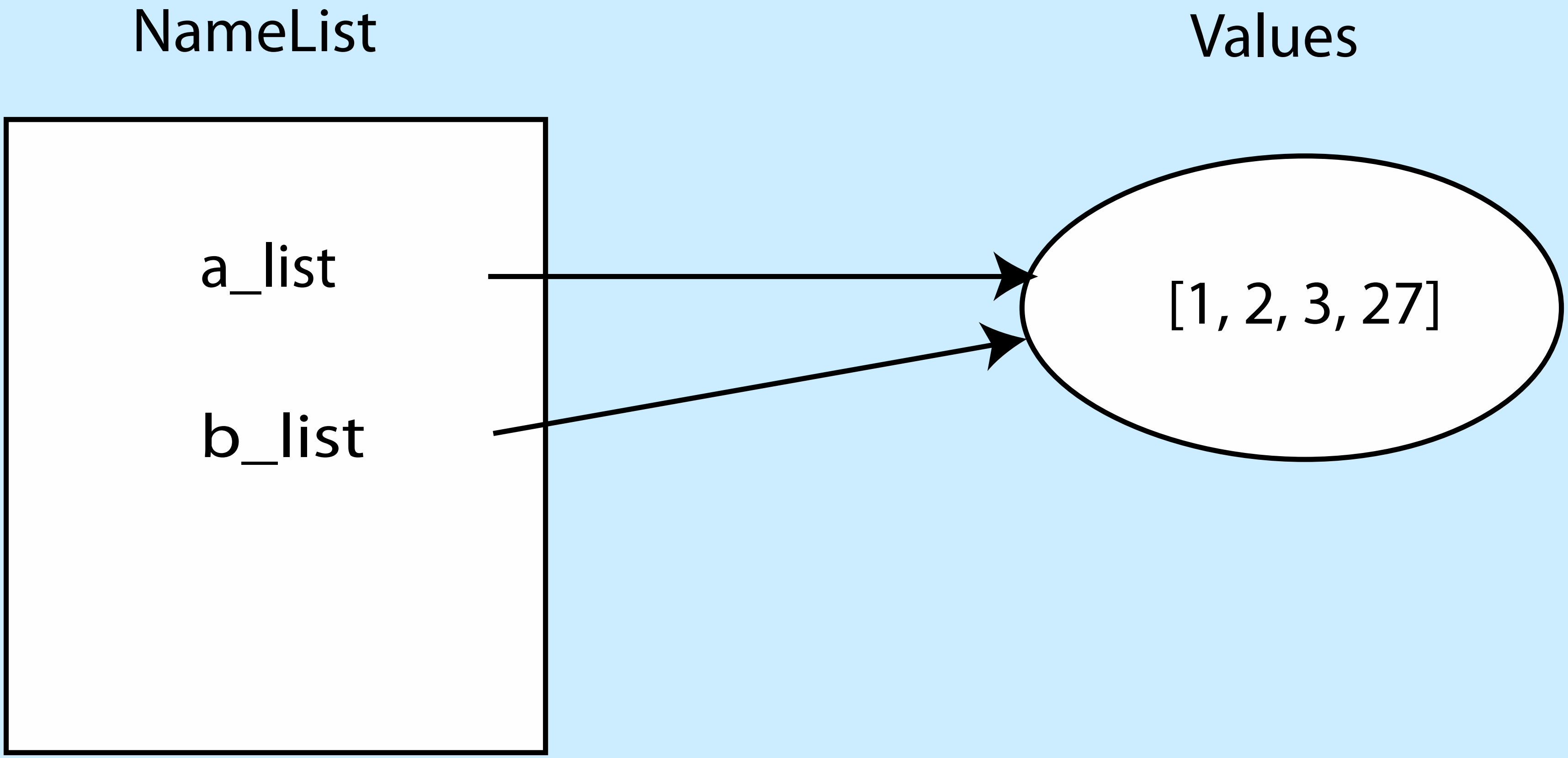
a_list

b_list

[1, 2, 3]

FIGURE 7.4 Namespace snapshot after assigning mutable objects.

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



NameList

Values

a_list

b_list

[1, 2, 3, 27]

FIGURE 7.5 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)
```

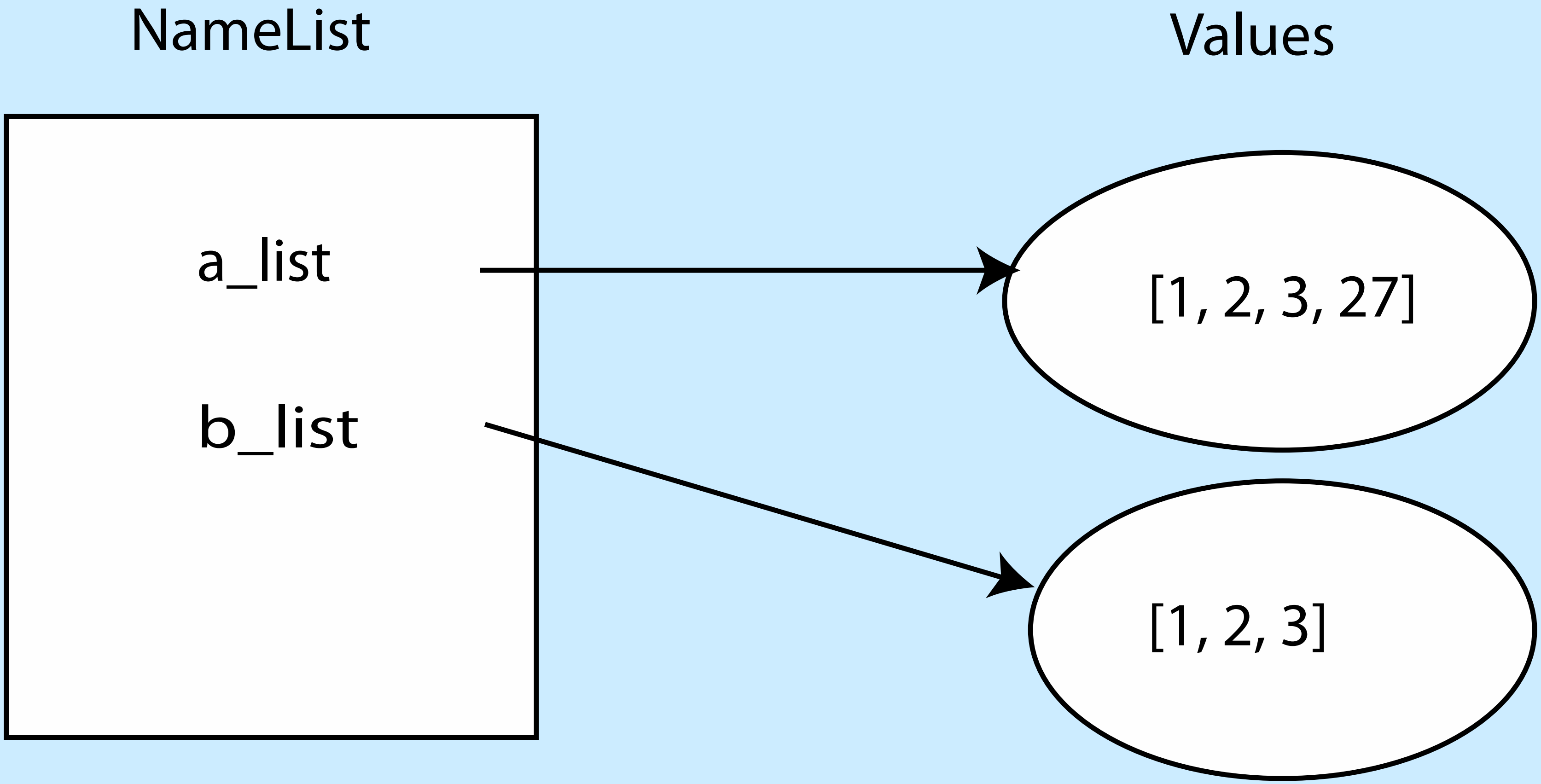


FIGURE 7.6 Making a distinct copy of a mutable object.

Sort_of/depends - what gets copied?

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, [...]]
```

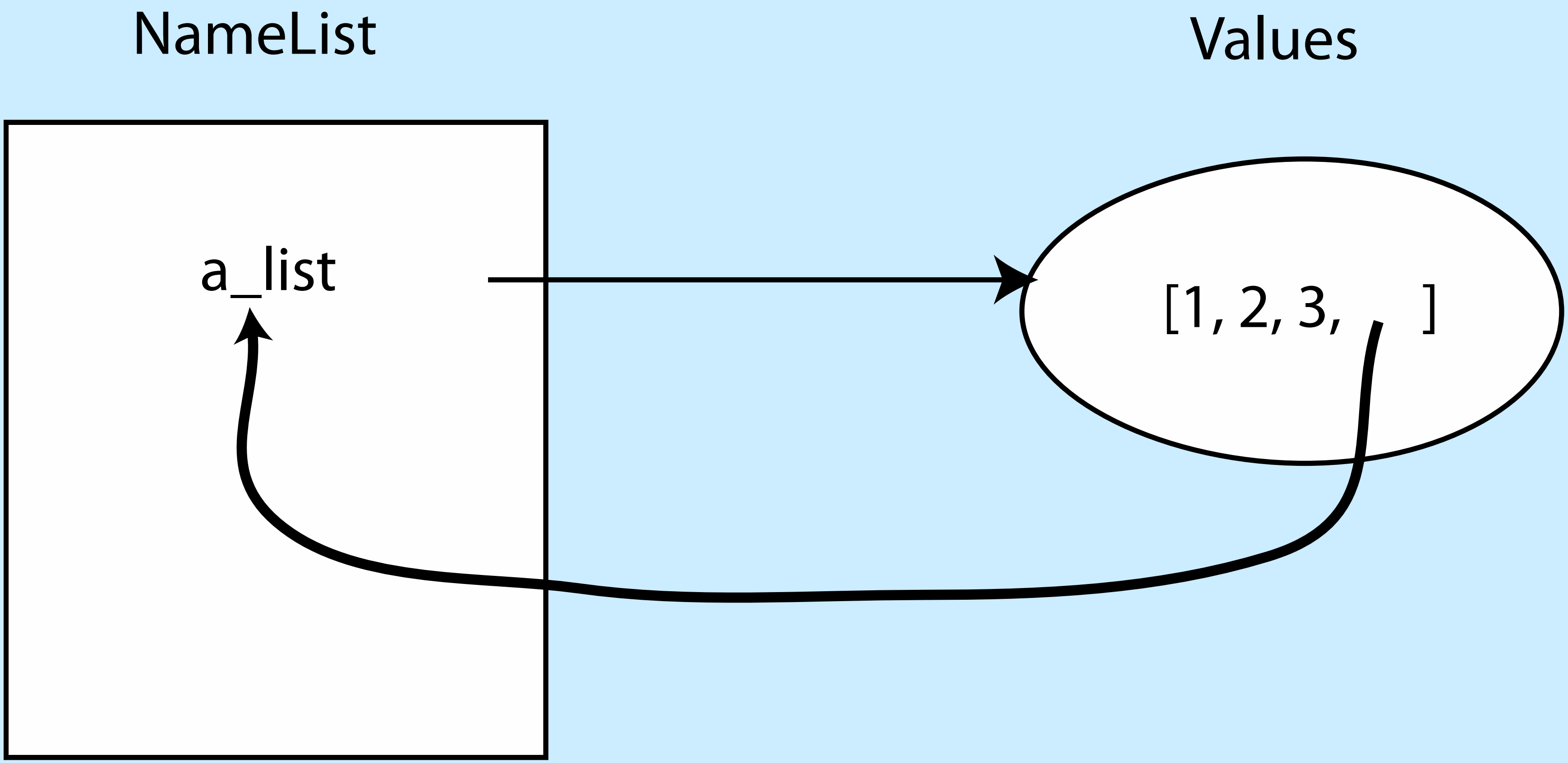


FIGURE 7.7 Self-referencing.

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

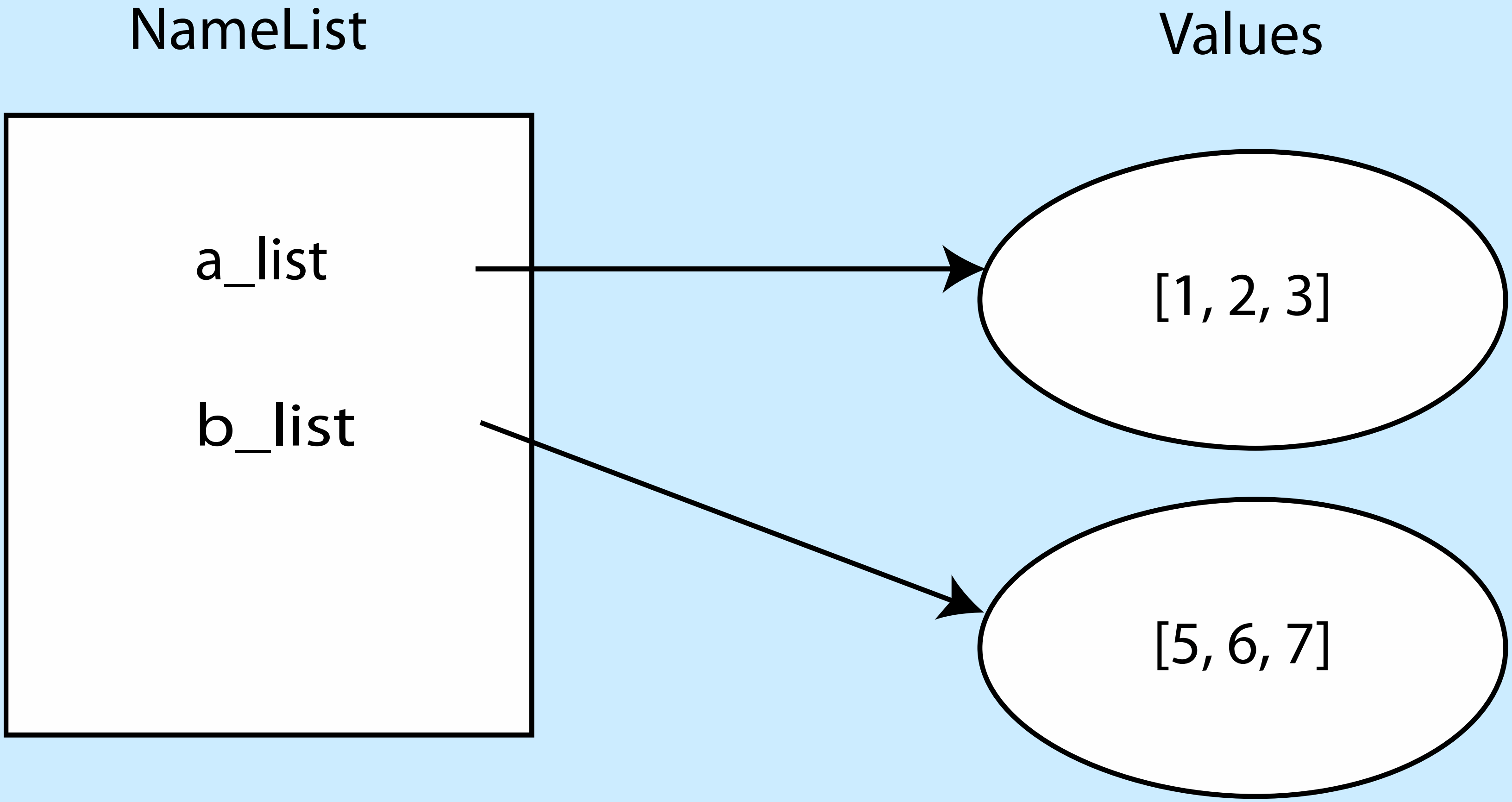
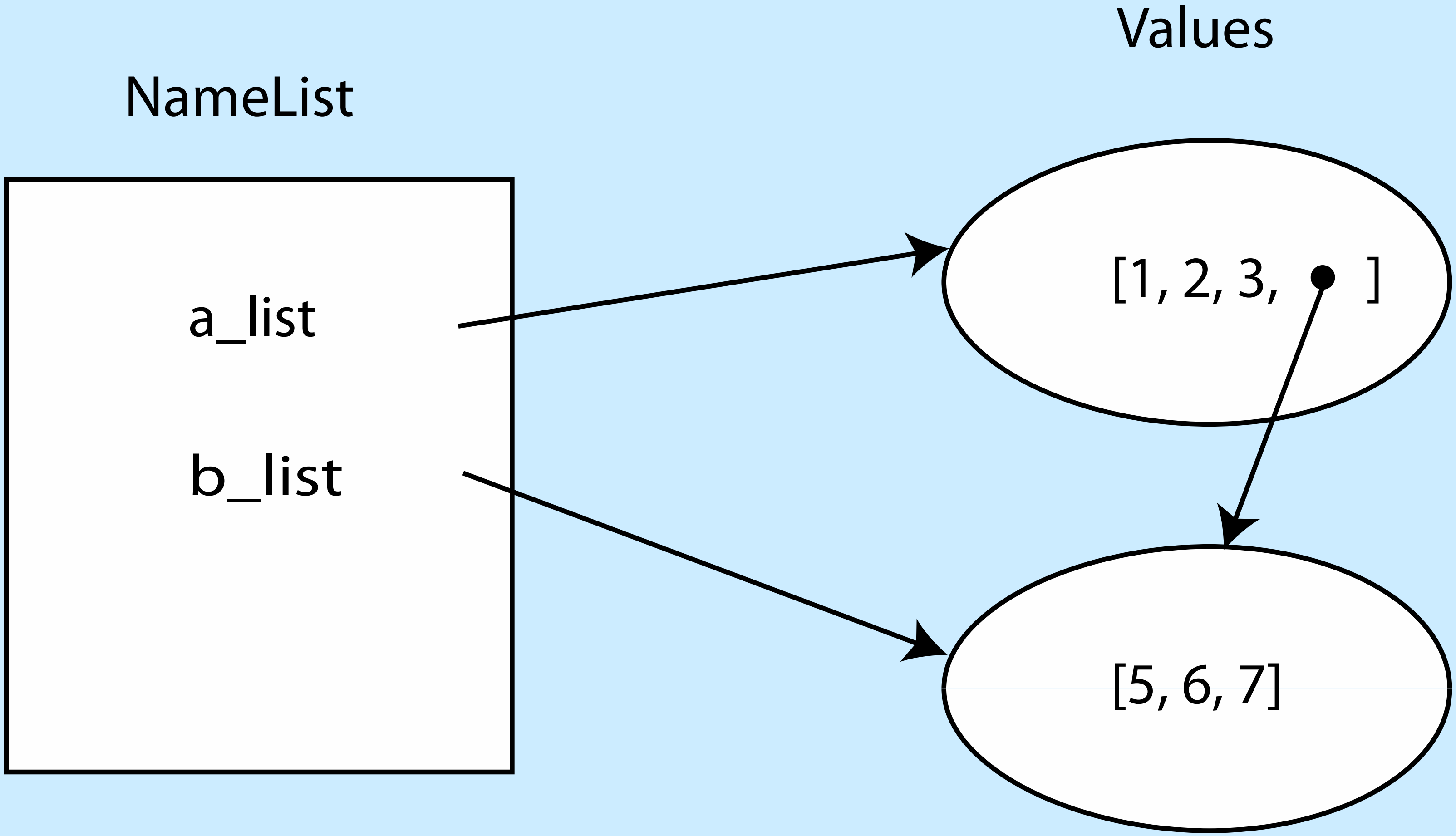


FIGURE 7.8 Simple lists before append.

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



Values

NameList

a_list

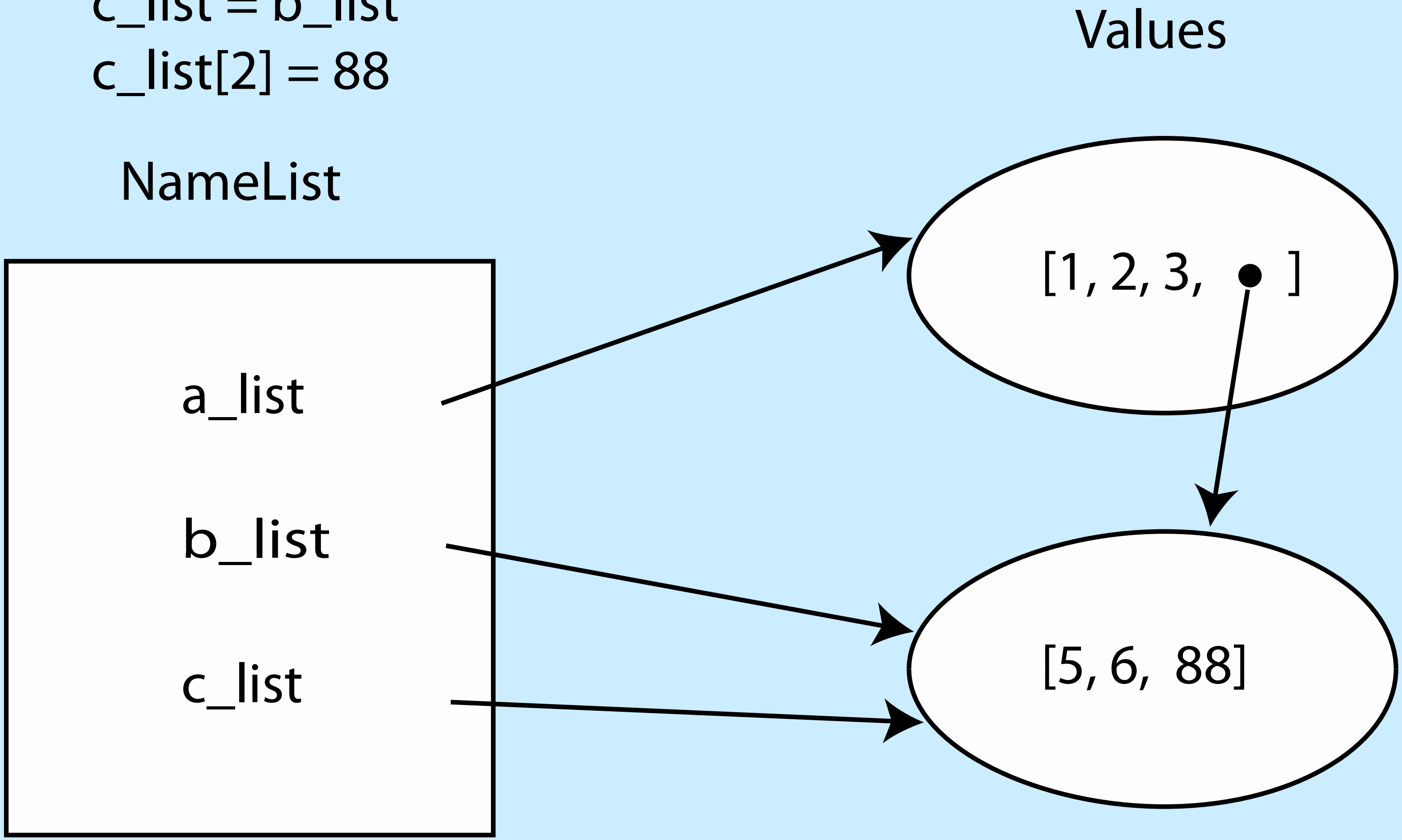
b_list

[1, 2, 3, ●]

[5, 6, 7]

FIGURE 7.9 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
```



Values

NameList

a_list

b_list

c_list

[1, 2, 3, ●]

[5, 6, 88]

FIGURE 7.10 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
```

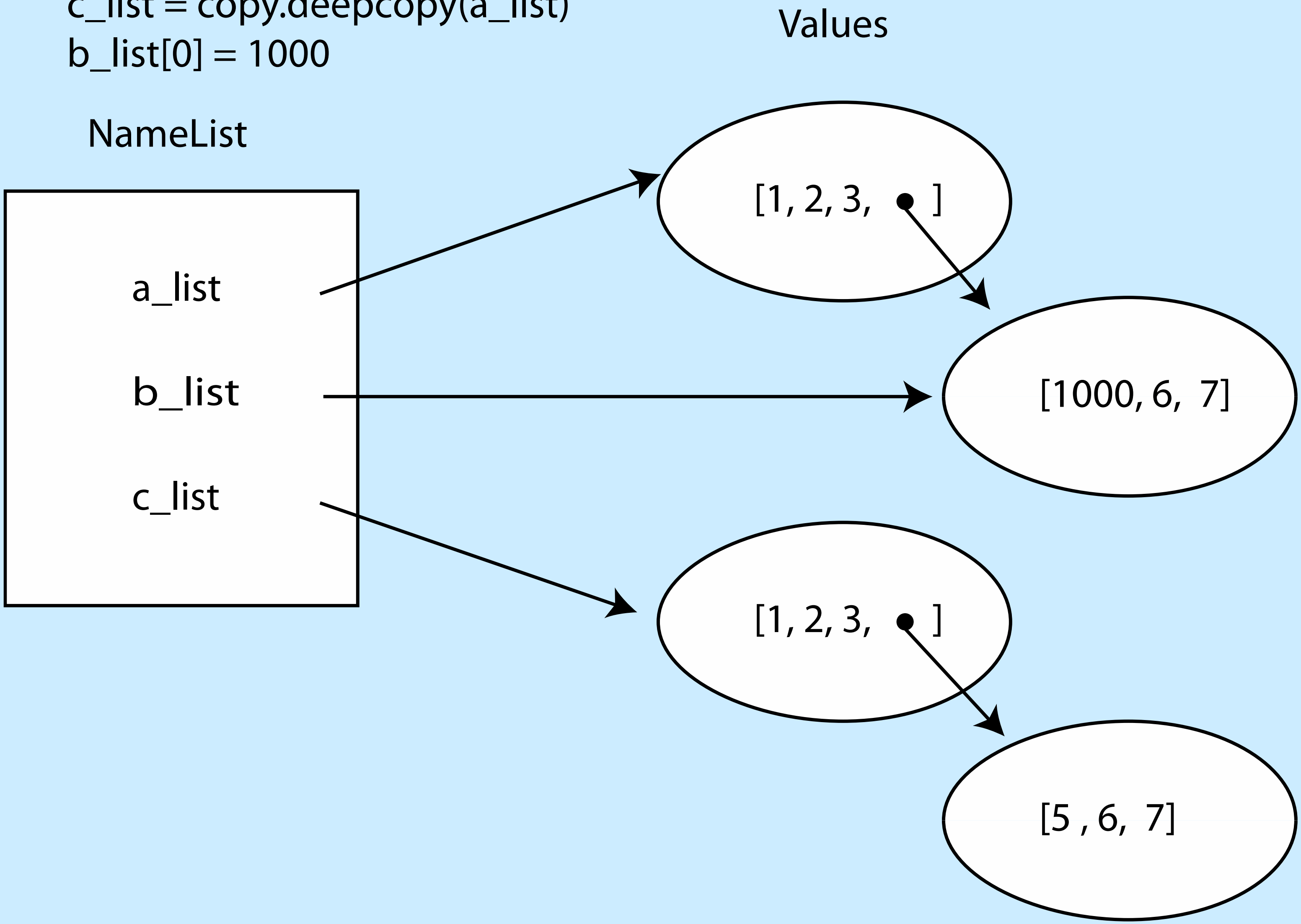


FIGURE 7.12 Using the `copy` module for a deep copy.

Tuples

Tuples

- Tuples are simply immutable lists
- They are printed with (,)

```
>>> 10,12            # Python creats a tuple
(10, 12)
>>> tup = 2,3        # assigning a tuple to a variable
>>> tup
(2, 3)
>>> (1)              # not a tuple, a grouping
1
>>> (1,)             # comma makes it a tuple
(1,)
>>> x,y = 'a',3.14159    # from on right, multiple assignments
>>> x
'a'
>>> y
3.14159
>>> x,y              # create a tuple
('a', 3.14159)
```

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 permanent-ness if you want
- You know you cannot accidentally change one.

List and Tuple

- 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 algorithm)
- Efficient with respect to the amount of space used
- Efficient with respect to the time it takes to perform some operations

List Comprehensions

Lists are a big deal!

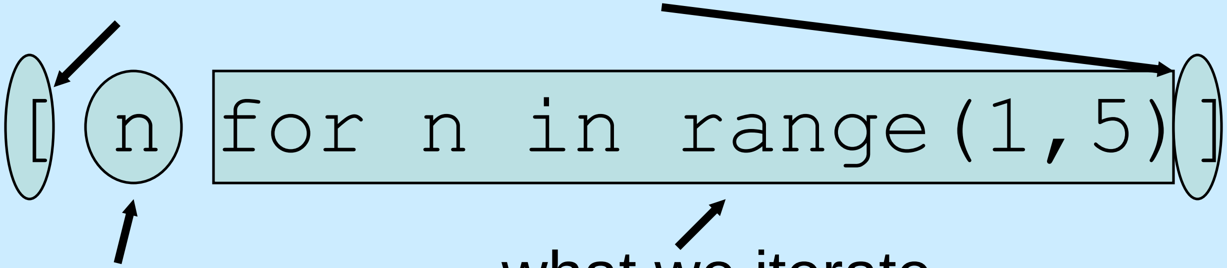
- The use of lists in Python is a major part of its power
- Lists are very useful and can be used to accomplish many tasks
- Therefore Python provides some pretty powerful support to make common list tasks easier

Constructing lists

One way is a "list comprehension"

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

mark the comprehension with [ ]



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

what we collect

what we iterate through. Note that we iterate over a set of values and collect some (in this case all) of them

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:

```
my_list = [ ]
for x in range (1,4):
  for y in range (1,4):
      my_list.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']