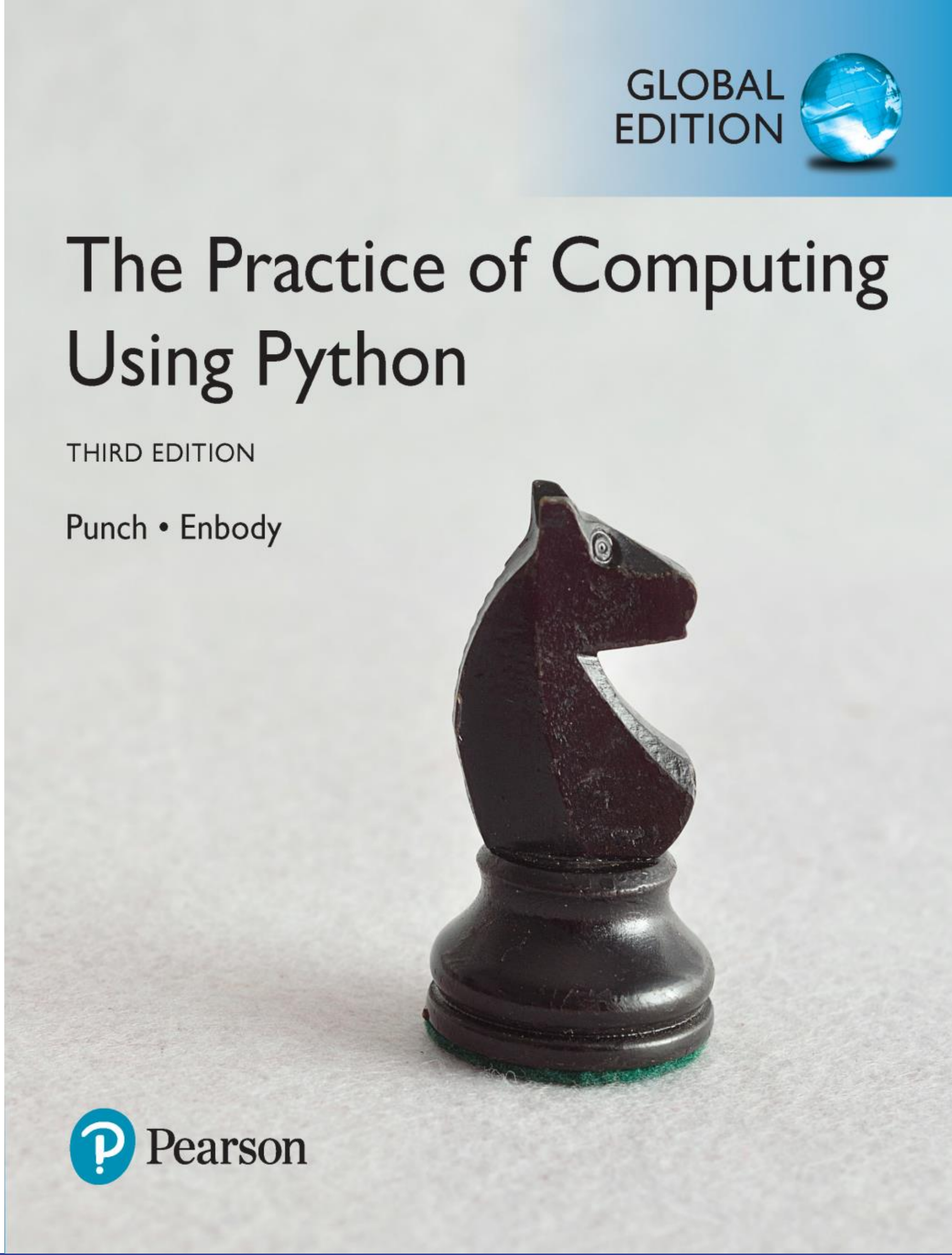
Lists and Tuples



GLOBAL EDITION

The Practice of Computing Using Python

THIRD EDITION

Punch • Enbody

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 topics

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 solve certain problems, and they are often associated with algorithms.

Kinds of data structures

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.

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 [ ] indicates explicit items.

List

```
a_list=[1,2,'a',3.14159]
week_day_list=['Monday','Tuesday','Wednesday','Thursday','Friday']
list_of_lists=[[1,2,3],['a','b','c']]
list_from_collection=list('Hello')
print(a_list)
print()
print(week_day_list)
print()
print(list_of_lists)
print()
print(list_from_collection)
```

```
[1, 2, 'a', 3.14159]

['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']

[[1, 2, 3], ['a', 'b', 'c']]

['H', 'e', 'l', 'l', 'o']
```

Similarities with strings

- Concatenate: +
- Repeat: *
- Indexing: [ ]
- Slicing: [:]
- Membership: the in operator
- Length: len()

Operators

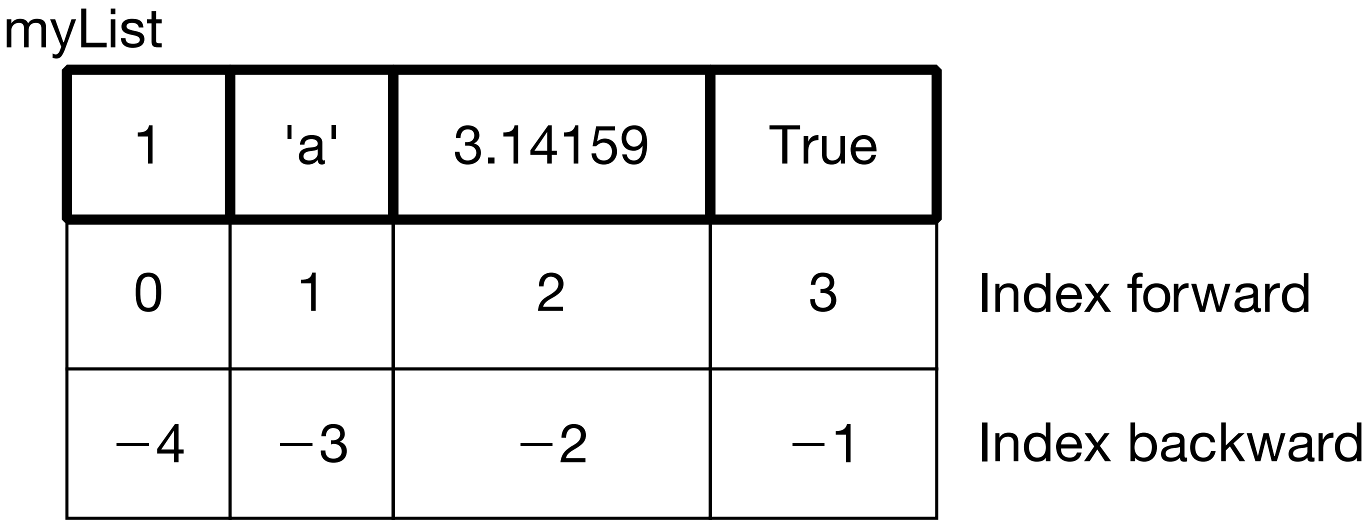
```
print([1,2,3]+[4])⇒ [1, 2, 3, 4]

print([1,2,3]*2) ⇒ [1, 2, 3, 1, 2, 3]

print(1 in [1,2,3]) ⇒ True

print([1,2,3]<[1,2,4])⇒ True
```

– compare index to index, the first difference determines the result

Differences between lists and strings

- lists can contain a mixture of any python object, strings can only hold characters
  - A=[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?

```
    print([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)

```
S="1234"
print(S [1])
L=['1','2','3','4']
print(L[1])
```

List of Lists

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

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

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

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

List Functions

- `len(lst)`: number of elements in list (top level).
  - print(len([1, [1, 2], 3])) ⇒ 3
- `min(lst)`: smallest element. Must all be the same type!
  - print(min([[100,2], [1,9200], [3,6]])) ⇒ `[1, 9200]`
- `max(lst)`: largest element. All elements must be the same type
  - print(max([[100,2,3], [200], [3,6]])) ⇒200
- `sum(lst)`: sum of the elements, numeric only
  - print(sum([1, 2, 3])) ⇒6

print(([sum([100,2,3]), [200], [3,6]]))

Iteration

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

```
my_list=[1,3,4,8]
for o in my_list:
    print(o, end=' ')
```

```
1 3 4 8
```

Simple Example

```
my_list = ['p', 'r', 'o', 'b', 'e']
print(my_list[0])
print(my_list[2])
print(my_list[4])
n_list = ["Happy", [2, 0, 1, 5]]
print(n_list[0][1])
print(n_list[1][3])
print(my_list[4.0])
```

Simple Example

```
my_list = ['p', 'r', 'o', 'b', 'e']
print(my_list[0]) # Output: p
print(my_list[2]) # Output: o
print(my_list[4]) # Output: e
n_list = ["Happy", [2, 0, 1, 5]] # Nested List
print(n_list[0][1]) # Output: a
print(n_list[1][3]) # Output: 5
print(my_list[4.0]) # Error! Only integer can be used for indexing
```

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

```
my_str = 'abac'
my_str[0] = 'z'
# instead, make new str
new_str = my_str.replace('a','z')
```

```
TypeError: 'str' object does not
support item assignment
```

```
my_str = 'abac'
new_str = my_str.replace('a','z')
print(new_str)
```

```
zbzc
```

Lists are mutable

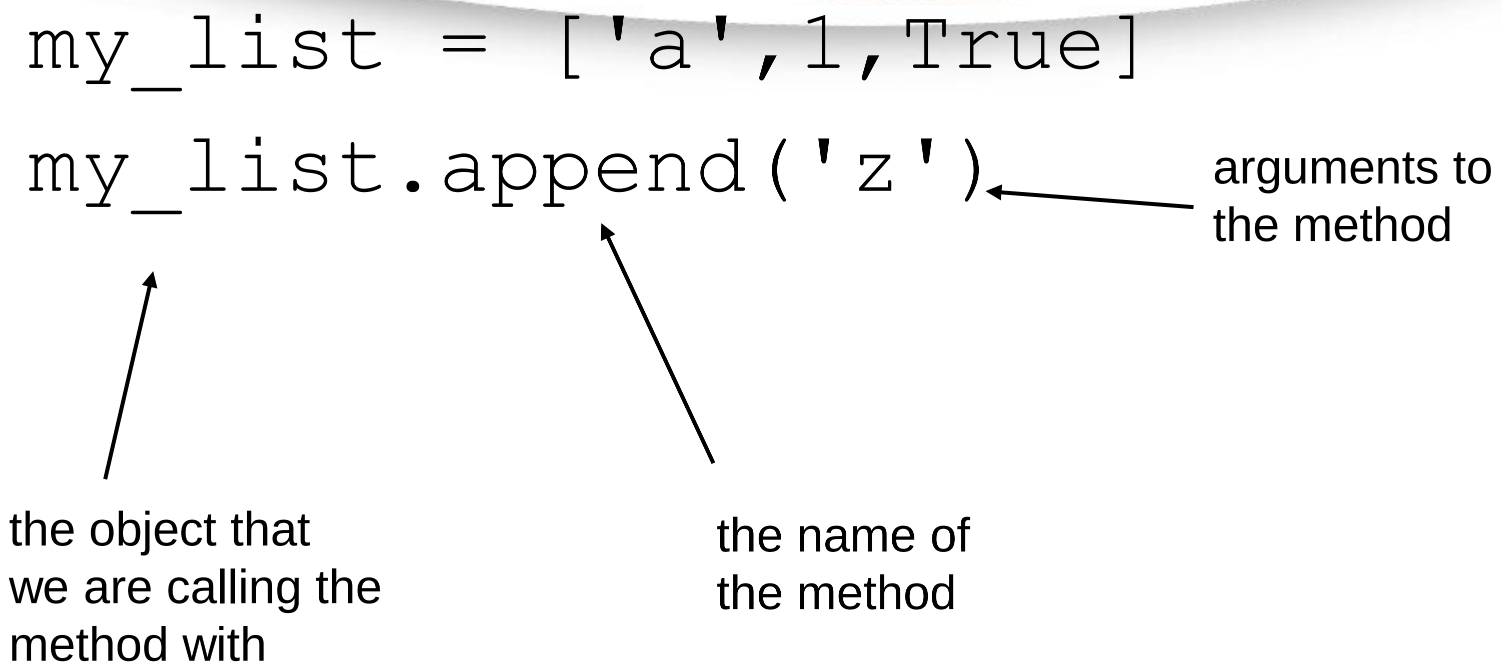
Unlike strings, lists are mutable. You ***can*** change the object's contents!

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

List methods

- Remember, a function is a small program (such as len()) 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

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

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

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.
- Can be confusing

Unusual results

```
my_list = [4, 7, 1, 2]
my_list = my_list.sort()
print(my_list) # 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.

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

```
[4, 7, 1, 2]
[1, 2, 4, 7]
```

Split

- The string method split generates a sequence of characters by splitting the string at certain split-characters.
- ***It returns a list*** (we mention that before)

```
split_list = 'this is a test'.split()
print(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')
print(my_list)
  ⇒['x','y','z','a','b','c']
my_list.sort()   # no return
print(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()
print(string_elements)
reversed_elements=[]
for o in string_elements:
    reversed_elements.append(o[::-1])
print(reversed_elements)
new_str=' '.join(reversed_elements)
print(new_str)
```

```
my_str='This is a test'
string_elements=my_str.split()
print(string_elements)
reversed_elements=[]
for o in string_elements:
    reversed_elements.append(o[::-1])
print(reversed_elements)
for o in reversed_elements:
    print(o,end=‘=')
```

```
['This', 'is', 'a', 'test']
['sihT', 'si', 'a', 'tset']
sihT si a tset
```

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')
print(sort_list)
```

⇒ `[' ','h','i','m','m','o']`

Practices

2022-05-06

Practice 20230922

```
my_list = ['az',1,True]
print(my_list)

print(my_list)
```

```
['az', 1, True]
['aaz', 'az', 1, True, 'aaz']
```

```
my_str = "az 1 True"
print(my_str)

print(my_str)
```

```
az 1 True
az 1 az True
```

Practice 20230922

```
my_string = 'az 1 True'

print(my_string)
```

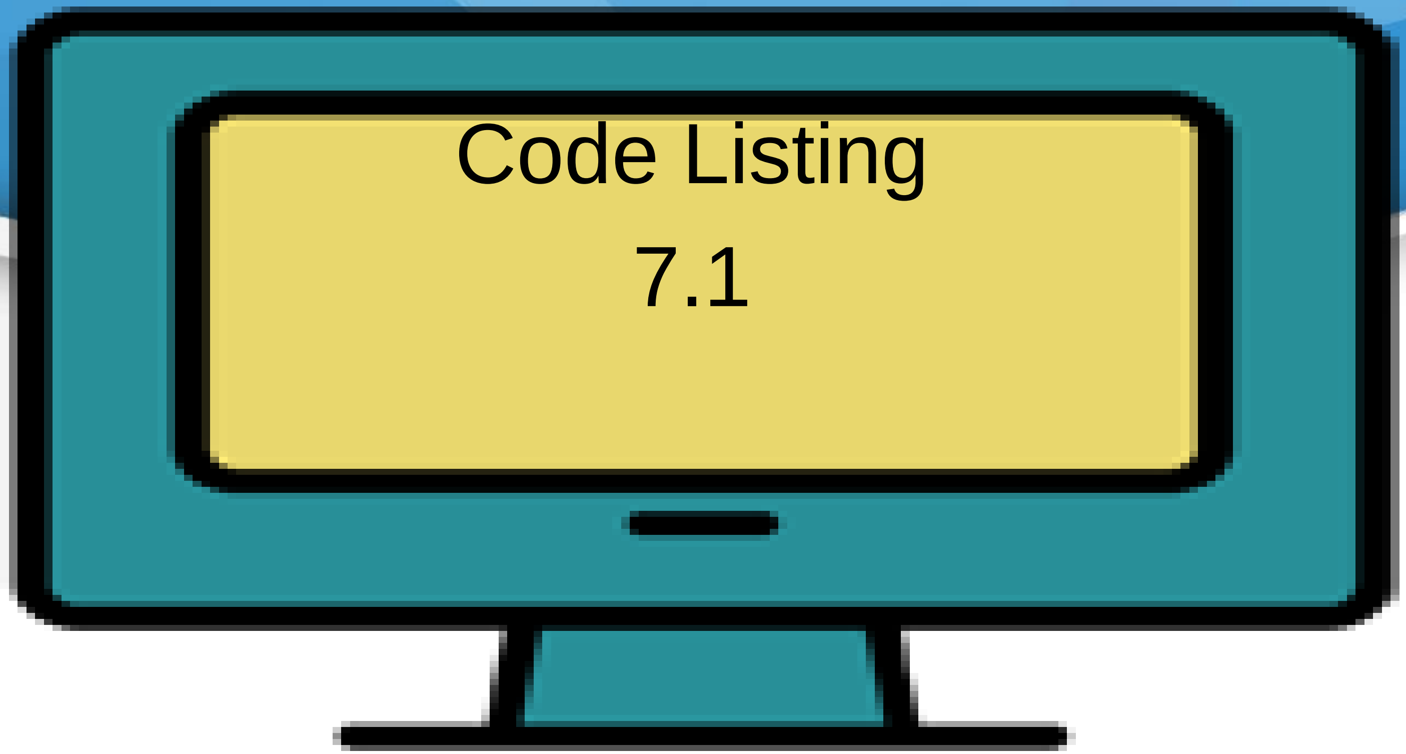
aaz az 1 True aaz

```
my_string = 'az 1 True'

print(my_string)
```

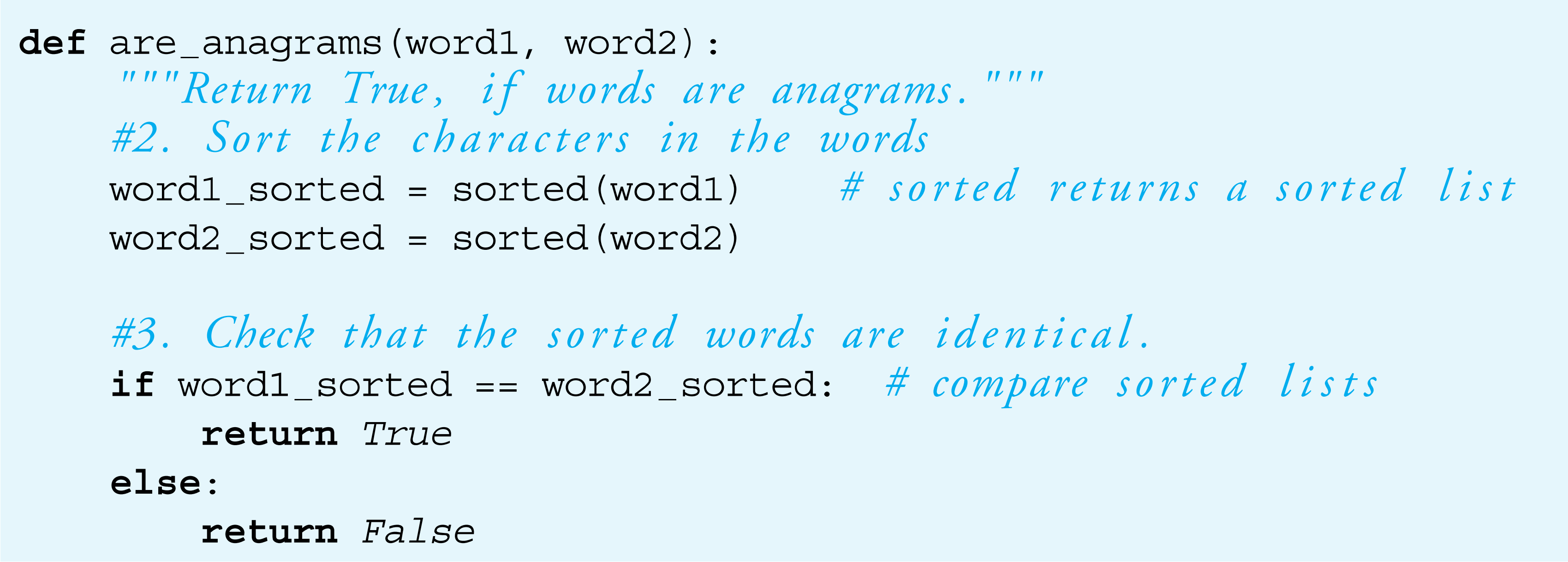
az 1 az True

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, than compare the sorted sequences. Anagrams should have the same sorted sequence

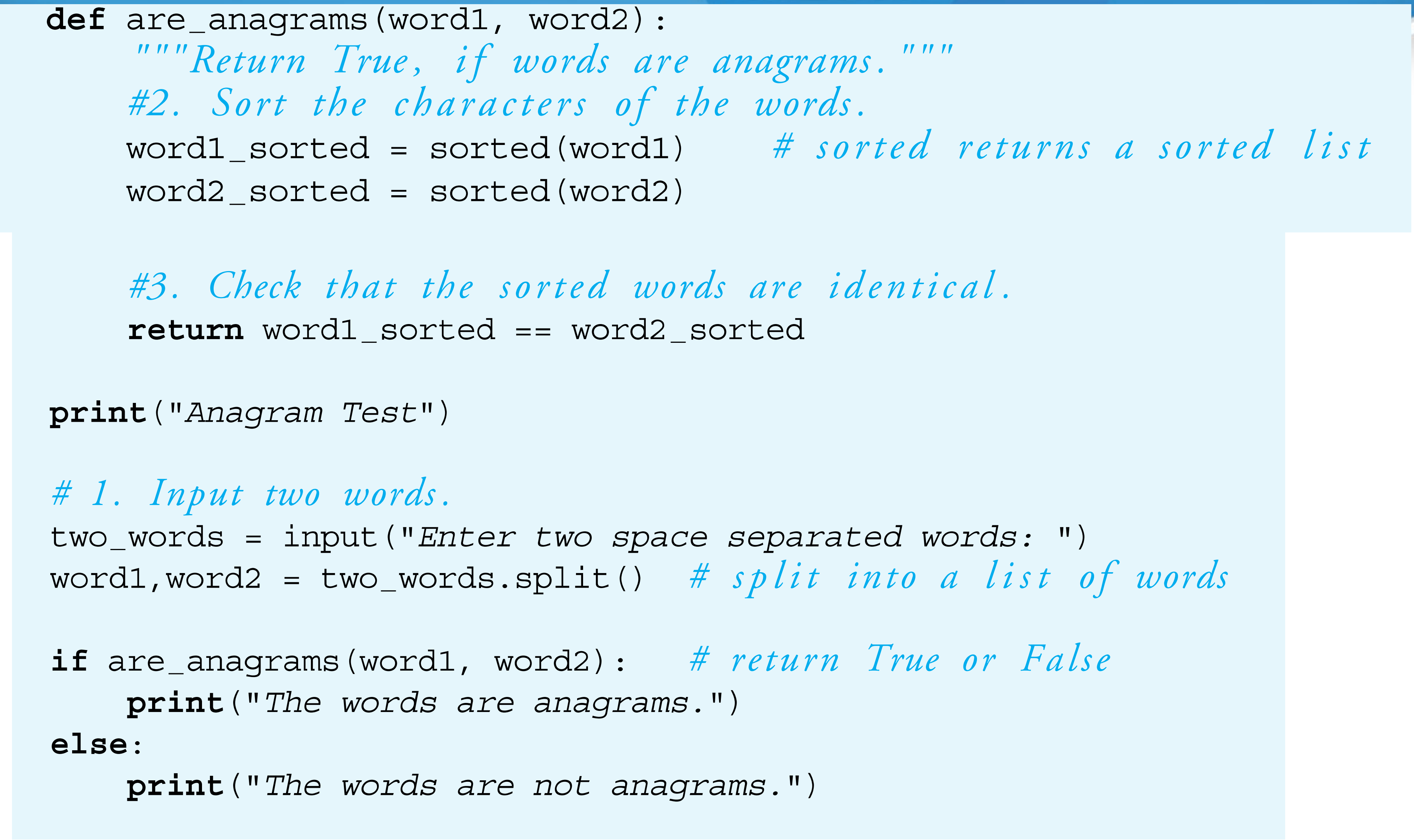


Code Listing
7.1

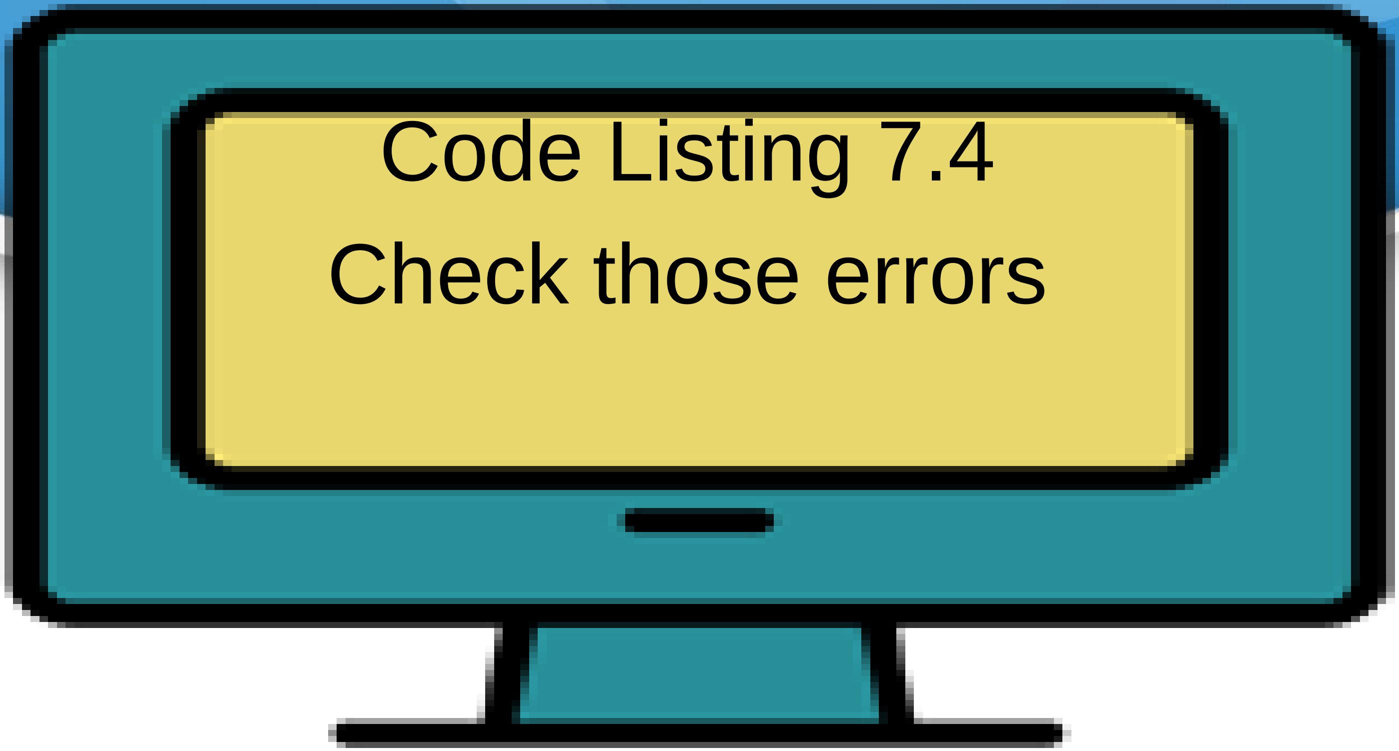


```
def are_anagrams(word1, word2):
    """Return True, if words are anagrams."""
    #2. Sort the characters in the words
    word1_sorted = sorted(word1)     # sorted returns a sorted list
    word2_sorted = sorted(word2)

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

Code Listing 7.3
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 7.4

Check those 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))
```

More about Mutables

2023-09-22

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

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

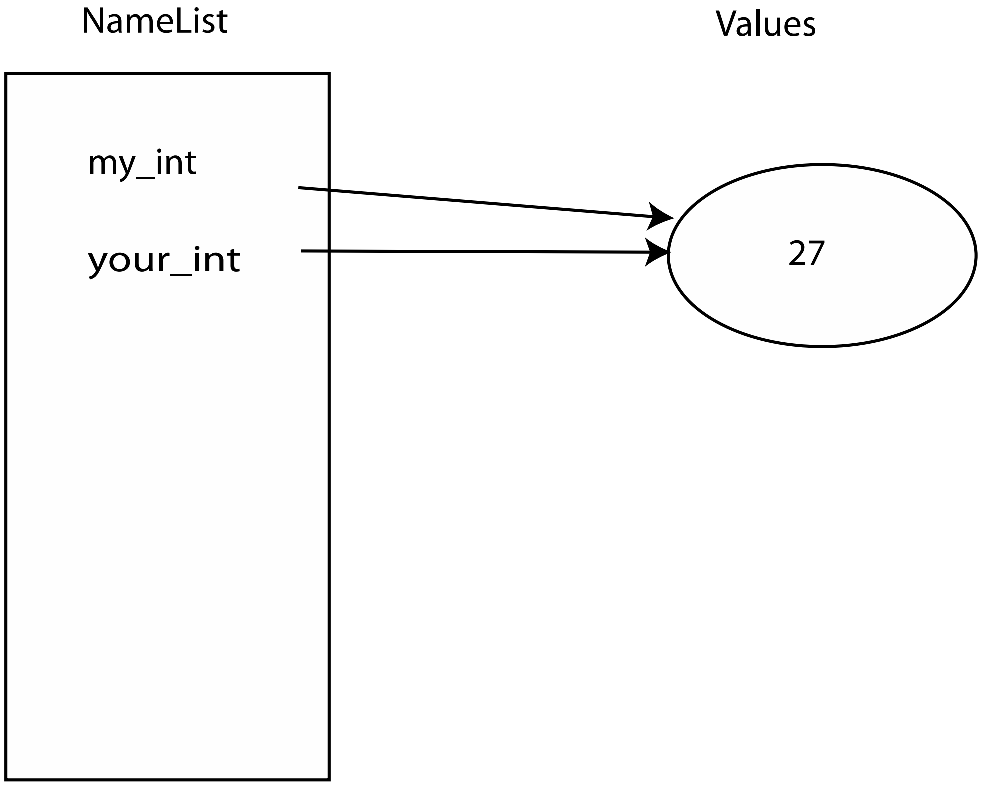


FIGURE 7.2 Namespace snapshot #1.

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
```

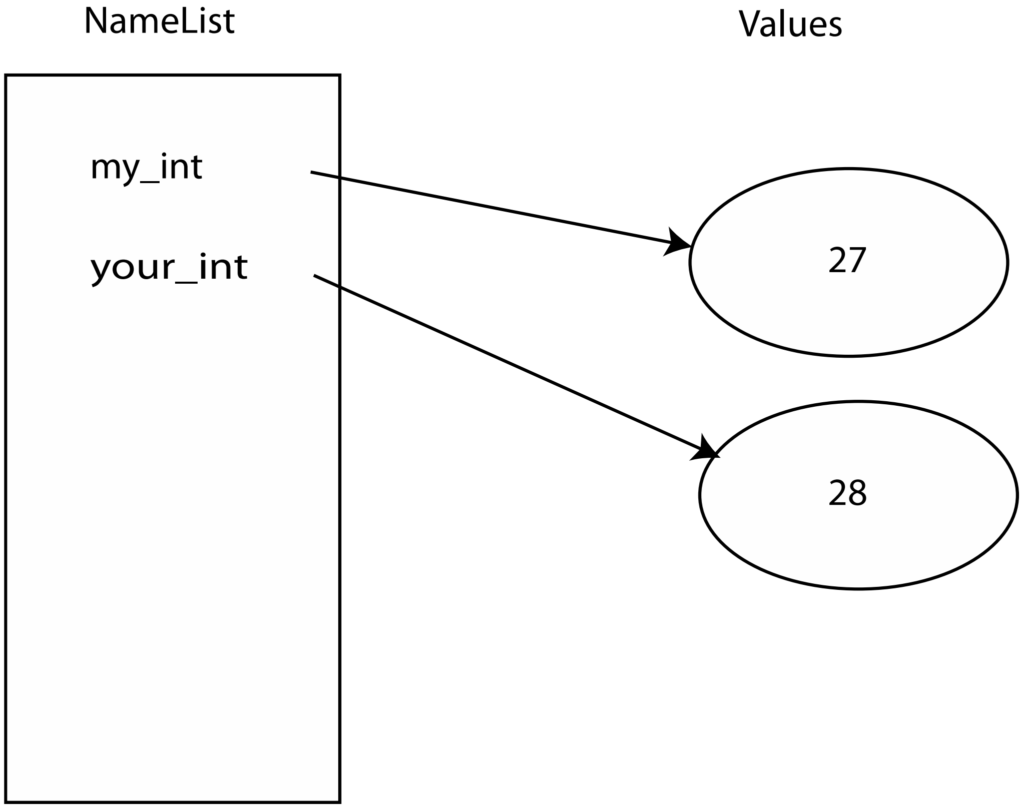


FIGURE 7.3 Modification of a reference to an immutable object.

Example

```
a = 2
b = 2
print(id(a),id(b))
a+=1
b+=1
print(id(a),id(b))
```

```
140723651580768 140723651580768
140723651580800 140723651580800
```

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
```

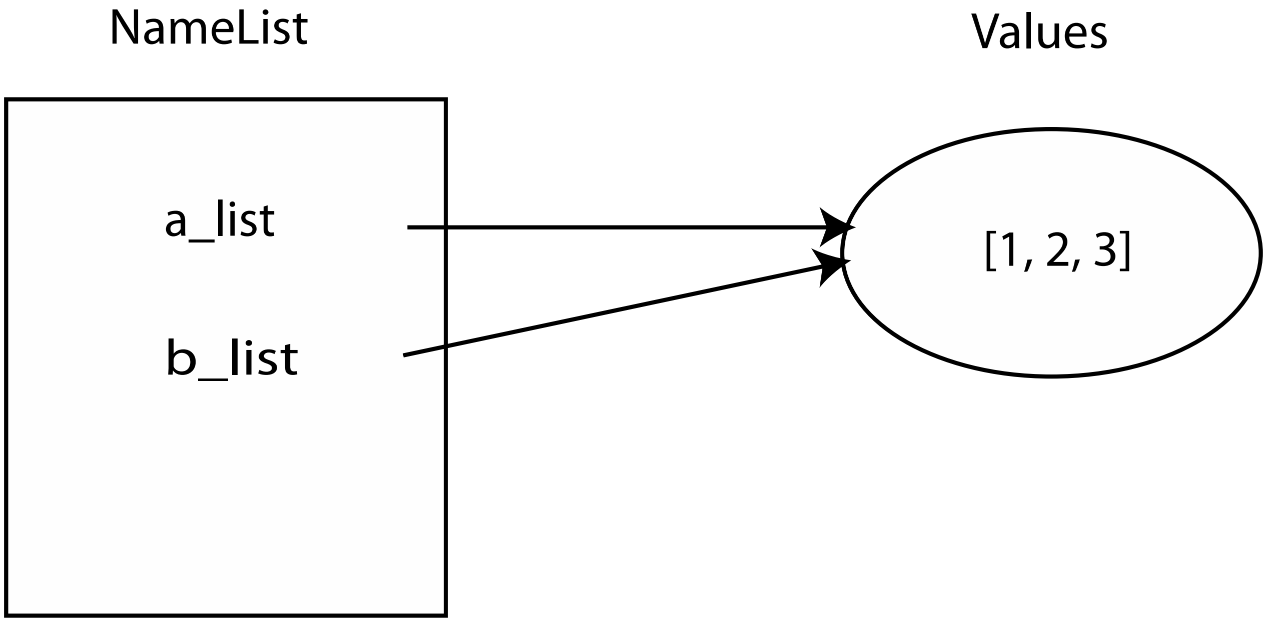


FIGURE 7.4 Namespace snapshot after assigning mutable objects.

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

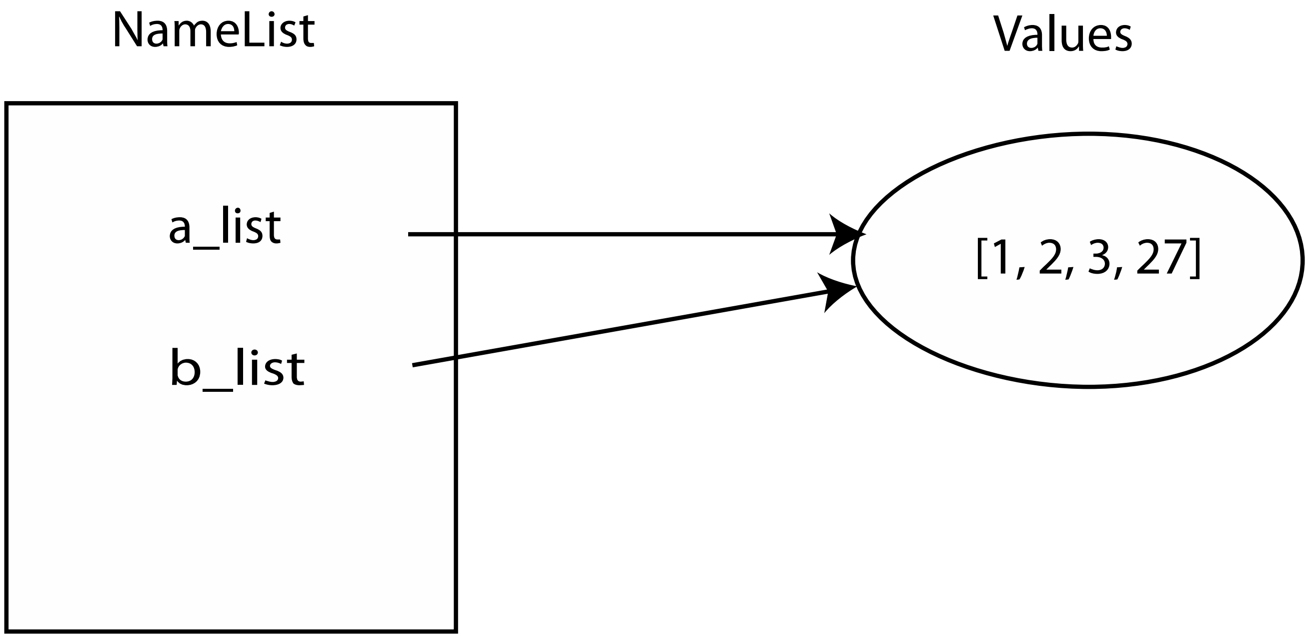


FIGURE 7.5 Modification of shared, mutable objects.

Example

```
alist=[1,2,3]
blist=alist
print(id(alist),id(blist))
alist[0]=9
alist.append('s')
print(id(alist),id(blist))
print(alist,blist)
```

```
1933268456136 1933268456136
1933268456136 1933268456136
[9, 2, 3, 's'] [9, 2, 3, 's']
```

Copying

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

```
my_list = [1, 2, 3]
newLst = my_list[:]
print(id(my_list))
print(id(newLst))
my_list.append(5)
newLst.append(4)
print(id(my_list))
print(id(newLst))
```

```
2025562480072
2025562062024
2025562480072
2025562062024
```

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

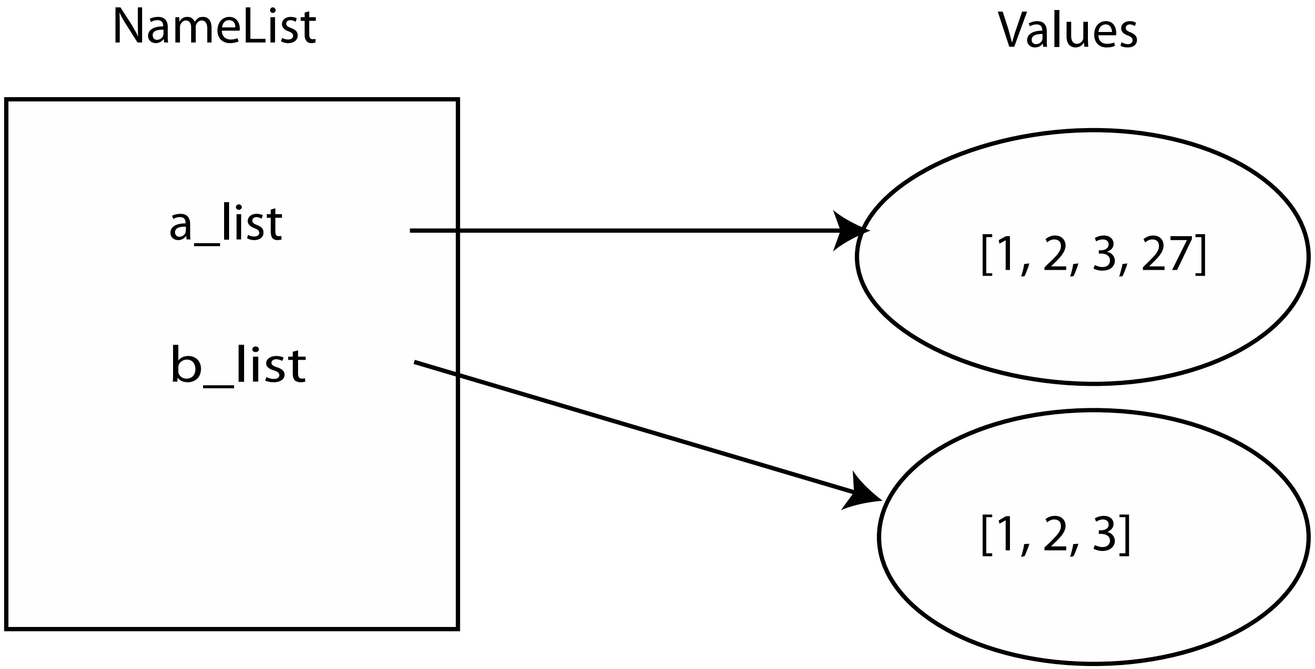


FIGURE 7.6 Making a distinct copy of a mutable object.

Example

```
alist=[1,2,3]
blist=alist[:]
print(id(alist),id(blist))
alist[0]=9
alist.append('s')
print(id(alist),id(blist))
print(alist, blist)
```

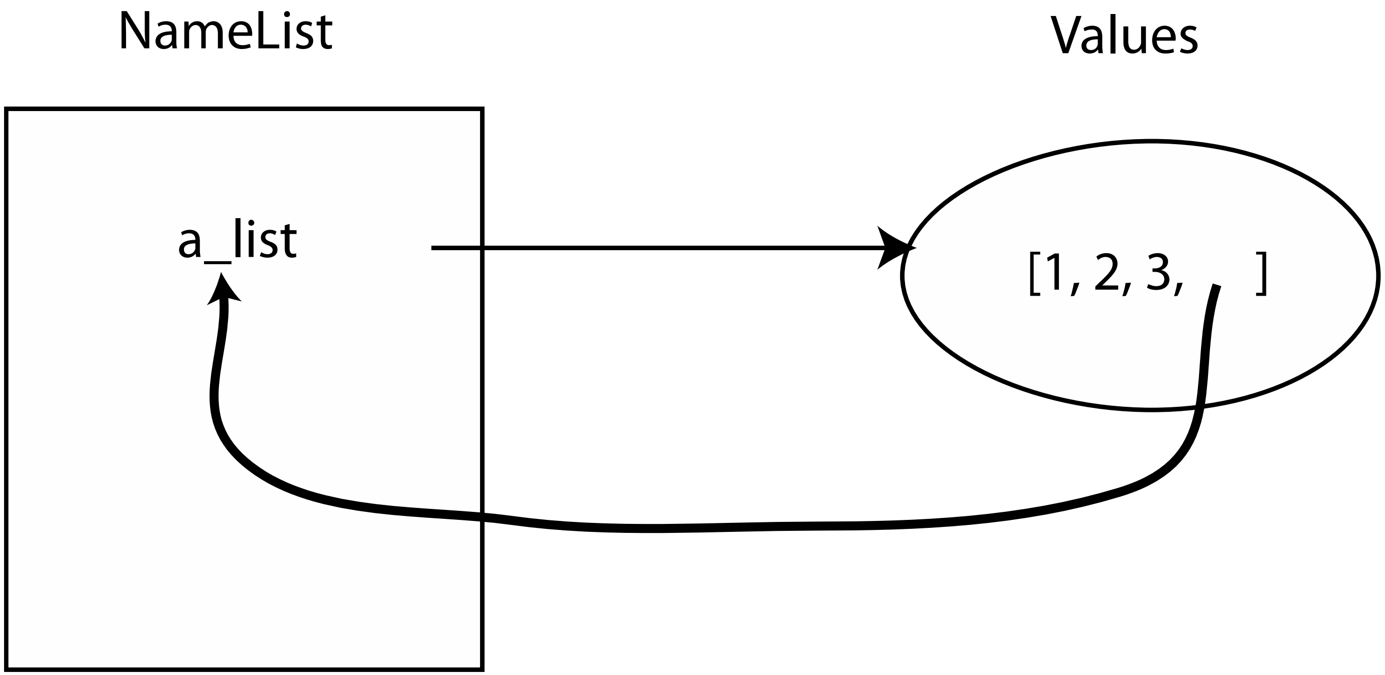
```
1933268488712 1933268491656
1933268488712 1933268491656
[9, 2, 3, 's'] [1, 2, 3]
```

Shallow Copy

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, [...]]
```



NameList

Values

a_list

[1, 2, 3,   ]

FIGURE 7.7 Self-referencing.

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

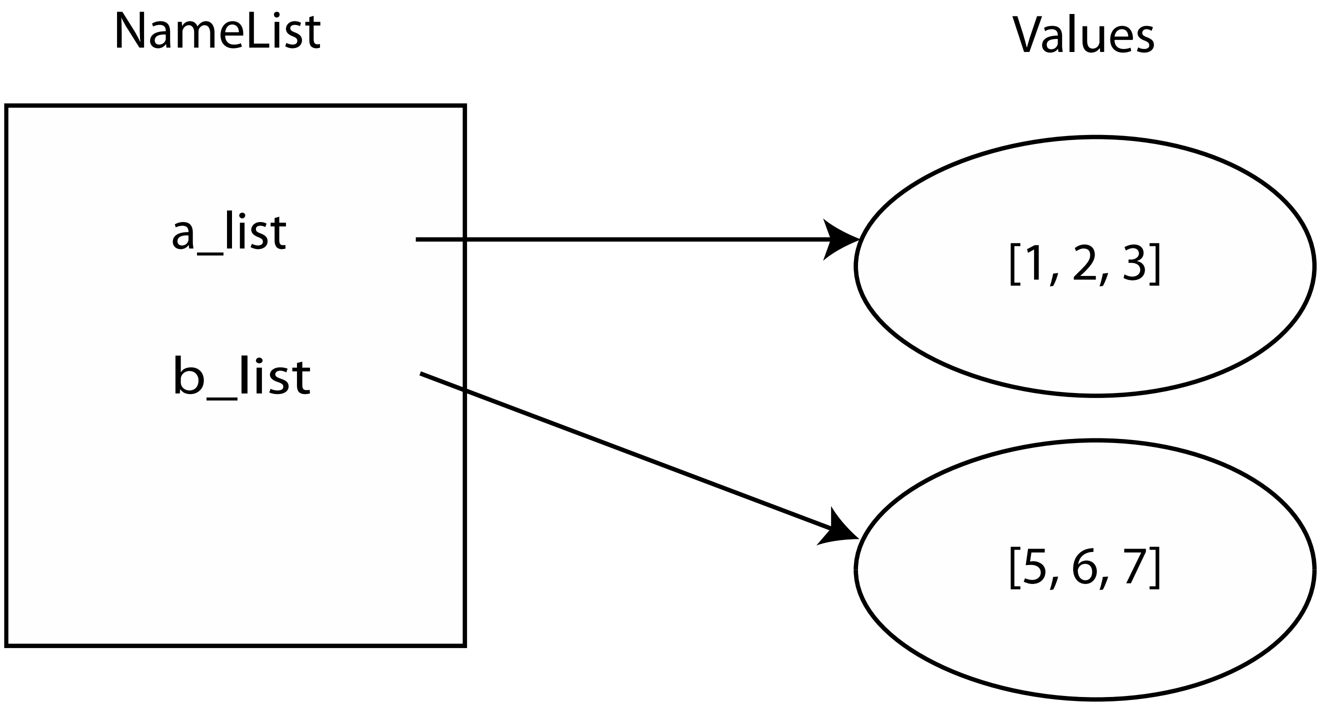


FIGURE 7.8 Simple lists before append.

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

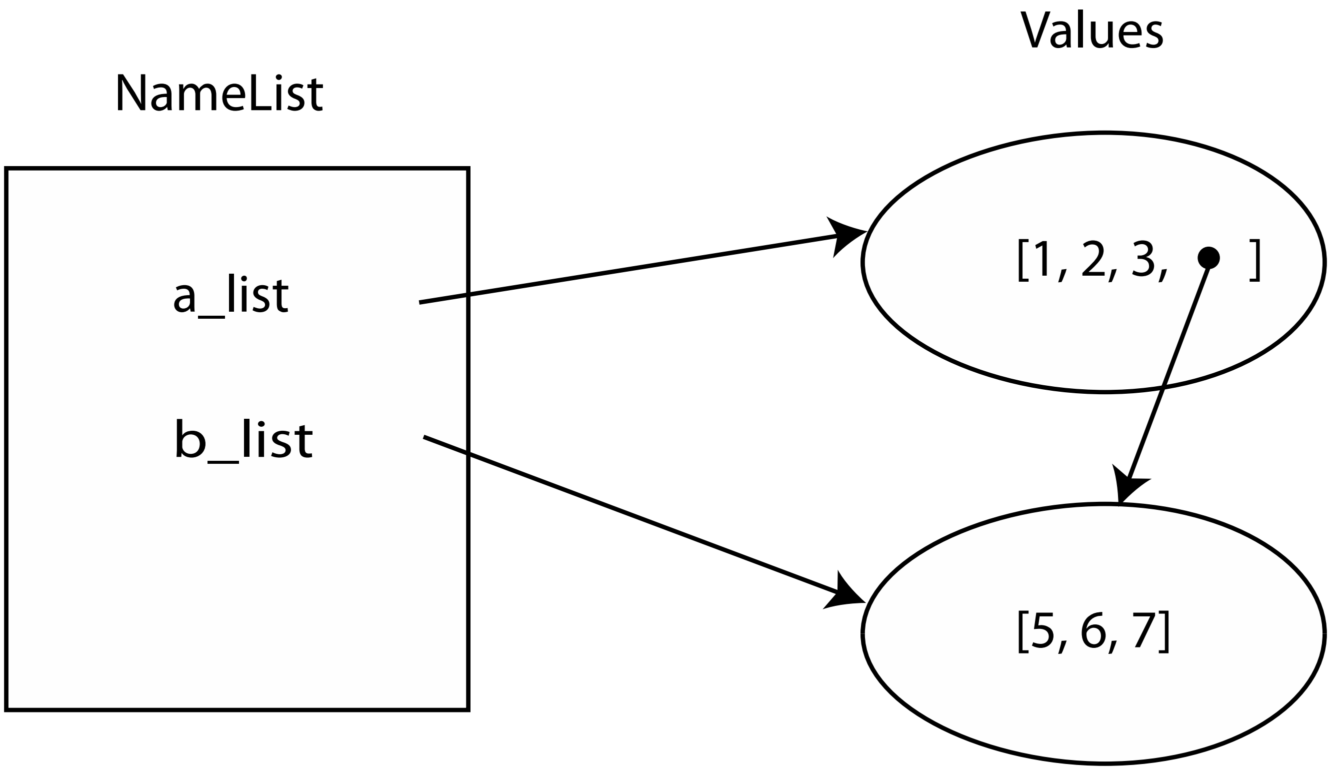
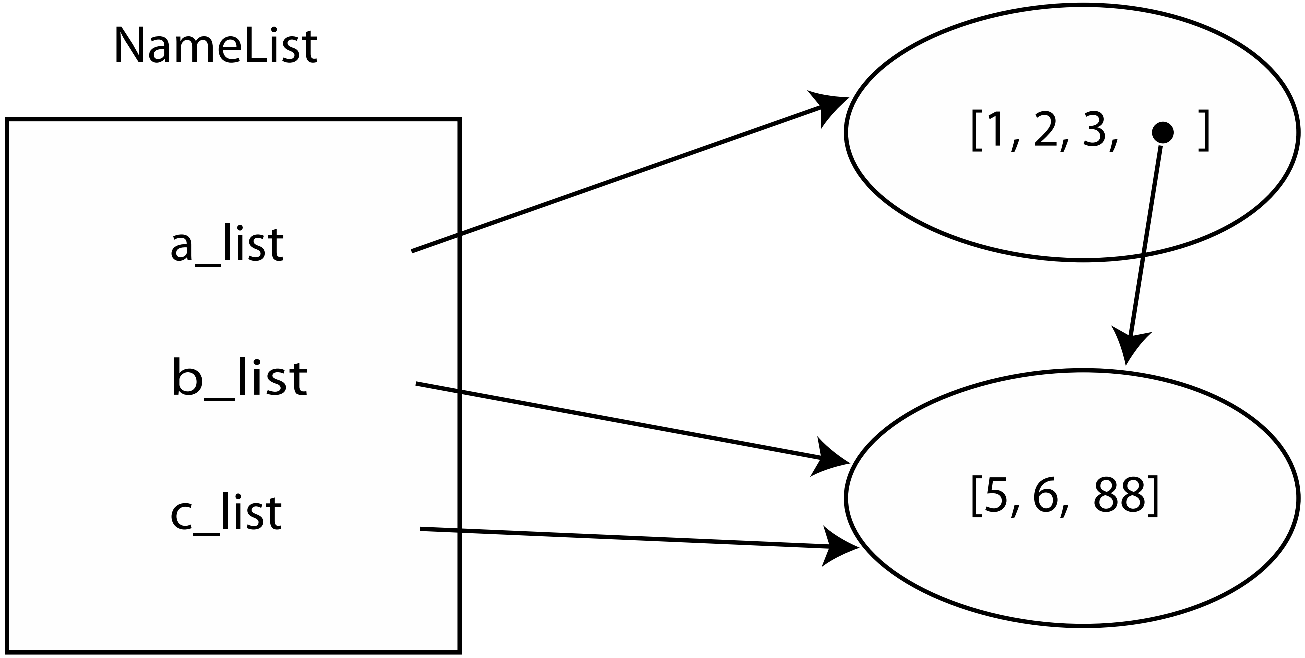


FIGURE 7.9 Lists after append.

```
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)
print(a_list)
import copy
c_list=copy.deepcopy(a_list)
print(c_list)
b_list[0]=1000
print(a_list)
print(c_list)
```

```
[1, 2, 3, [5, 6, 7]]
[1, 2, 3, [5, 6, 7]]
[1, 2, 3, [1000, 6, 7]]
[1, 2, 3, [5, 6, 7]]
```

Assignment, Shallow & Deep

```
import copy
O=[[1,2],3,4]
A=O
S=copy.copy(O)
D=copy.deepcopy(O)
print('O:\t',O,'\tO-ID: ',id(O),'\tO-ID[0]: ',id(O[0]))
print('A:\t',A,'\tA_ID: ',id(A),'\tA_ID[0]: ',id(A[0]))
print('S:\t',S,'\tS_ID: ',id(S),'\tS_ID[0]: ',id(S[0]))
print('D:\t',D,'\tD_ID: ',id(D),'\tD_ID[0]: ',id(D[0]))
```

```
O:      [[1, 2], 3, 4]      O-ID:  2572670520136    O-ID[0]:  2572670564296
A:      [[1, 2], 3, 4]      A_ID:  2572670520136    A_ID[0]:  2572670564296
S:      [[1, 2], 3, 4]      S_ID:  2572670565512    S_ID[0]:  2572670564296
D:      [[1, 2], 3, 4]      D_ID:  2572670603976    D_ID[0]:  2572670674376
```

append(), extend() & insert()

```
list_1 = ['object1', 'object2', 'object3']
list_2 = ['object4', 'object5']
list_1.extend(list_2)
print(list_1)
```

```
['object1', 'object2', 'object3', 'object4', 'object5']
```

```
list_1 = ['object1', 'object2', 'object3']
list_2 = ['object4', 'object5']
list_1.append(list_2)
print(list_1)
```

```
['object1', 'object2', 'object3', ['object4', 'object5']]
```

```
list_1 = ['object1', 'object2', 'object3']
list_2 = ['object4', 'object5']
list_1.insert(1,list_2)
print(list_1)
```

```
['object1', ['object4', 'object5'], 'object2', 'object3']
```

List Comprehensions

Lists are a big deal!

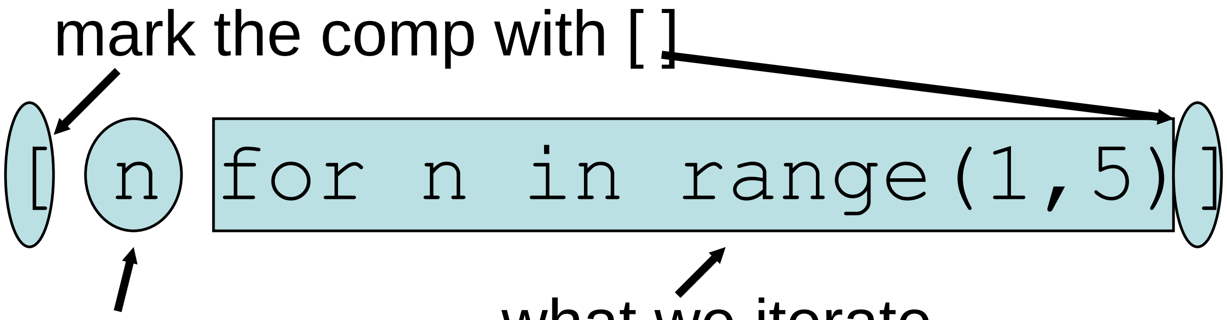
- The use of lists in Python is a major part of its power
- Lists are very useful and can be used to accomplish many tasks
- Therefore Python provides some pretty powerful support to make common list tasks easier

Constructing lists

One way is a "list comprehension"

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

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



mark the comp with [ ]

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

returns
[1,2,3,4]

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

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`

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

print(b)
```

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]

Exercise 1

- Unfinished code

```
my_list = ['*'*(x+y) for x in range (1,4) for y in range (1,4) ]
print(my_list)
```

- Result

```
['**', '***', '****', '***', '****', '*****', '****', '*****', '******']
```

Exercise 2

- Unfinished code

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

- Result

```
['**', '***', '****', '***', '****', '*****', '****', '*****', '******']
```

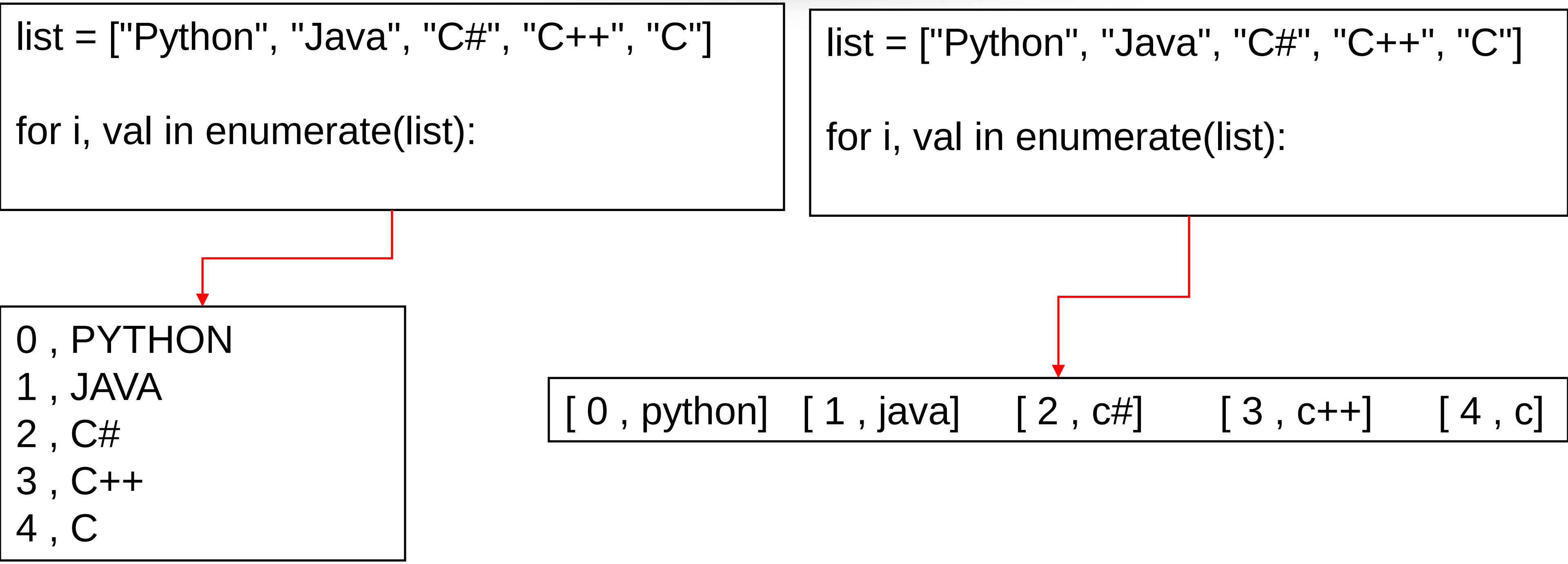
Enumerate() function

- If we want to convert the list into an iterable list of tuples (or get the index based on a condition check, for example in linear search you might need to save the index of minimum element), you can use the enumerate() function.

```
list = ["Python", "Java", "C#", "C++", "C"]

for i, val in enumerate(list):
    print (i, ",",val)
```

```
0 , Python
1 , Java
2 , C#
3 , C++
4 , C
```

Exercise 3



```
list = ["Python", "Java", "C#", "C++", "C"]

for i, val in enumerate(list):
```

```
0 , PYTHON
1 , JAVA
2 , C#
3 , C++
4 , C
```

```
list = ["Python", "Java", "C#", "C++", "C"]

for i, val in enumerate(list):
```

```
[ 0 , python]   [ 1 , java]     [ 2 , c#]       [ 3 , c++]       [ 4 , c]
```

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']

Exercise 4

```
list = [48, 65, 98]
for o in list:
    print(                    )
```

```
0      A      b
```

```
mylist=[c for c in "Hi 123 There 789 Mom 0"                                    ]
print(mylist)
```

```
['H', 'i', 'T', 'h', 'e', 'r', 'e', 'M', 'o', 'm']
```

Tuples

Tuple

- A tuple is a collection which is ordered and **unchangeable**. In Python tuples are written with round brackets, i.e., ().

Tuples

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

```
tup=2,3 #assigning a tuple to a variable
print(tup)
print((1,)) #comma makes it a tuple
x,y='a',3.14159 # multiple assignments
z=(x,y) #assigning a tuple to a variable
print(z)
```

Tuples

- Create a Tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

- Access Tuple Items
  - you can access tuple items by referring to the index number

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

Tuples

- Negative indexing

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

- Range of Indexes

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```

Tuples

- Change Tuple Values
  - 'tuple' object does not support item assignment

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

Tuples

- Loop Through a Tuple

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

- Check if Item Exists

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

Tuples

- Tuple Length

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

- Join Two Tuples

```
tuple1 = ("a", "b", "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3)
```

Tuples

- The tuple() Constructor: using the tuple() method to make a tuple.

```
thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
print(thistuple)
```

- The method of count()
  - Return the number of times the value 5 appears in the tuple

```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = thistuple.count(5)
print(x)
```

Tuples

- The method of index()
  - Search for the first occurrence of the value 8, and return its position:

```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = thistuple.index(8)
print(x)
```

Exercise 20230922

- Search for all occurrences of the value 8, and return their positions:

```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
```

Tuples

- Add Items
  - You cannot add items to a tuple:

```
thistuple = ("apple", "banana", "cherry")
thistuple[3] = "orange" # This will raise an error
print(thistuple)
```

- Remove Items
  - **Note:** You cannot remove items in a tuple.

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

Exercise

- Finish the program to add an item in a tuple.

```
tuplex = (4, 6, 2, 8, 3, 1)
print(tuplex)
#tuples are immutable, so you can not add new elements
#using merge of tuples with the + operator you can add an element and it will create a new tuple
tuplex = tuplex + (9,)
print(tuplex)
#adding items in a specific index
tuplex = tuplex[:5] + (15, 20, 25) + tuplex[:5]
print(tuplex)
#converting the tuple to list
listx = list(tuplex)
#use different ways to add items in list
listx.append(30)
tuplex = tuple(listx)
print(tuplex)
```

```
(4, 6, 2, 8, 3, 1)
(4, 6, 2, 8, 3, 1, 9)
(4, 6, 2, 8, 3, 15, 20, 25, 4, 6, 2, 8, 3)
(4, 6, 2, 8, 3, 15, 20, 25, 4, 6, 2, 8, 3, 30)
```

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 will
- You know you cannot accidentally change one.

Lists 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     # <class 'tuple'>
myTuple = (1,)    # <class 'tuple'>
myTuple = (1)     # <class 'int'>
myTuple = 1,      # <class 'tuple'>
```

Data Structures in General

Organization of data

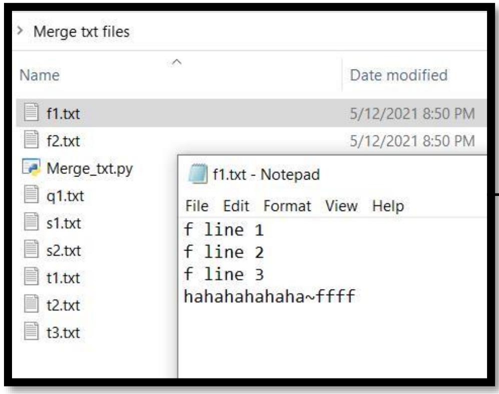
- We have seen strings, lists and tuples so far
- Each is an organization of data that is useful for some things, not as useful for others.

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

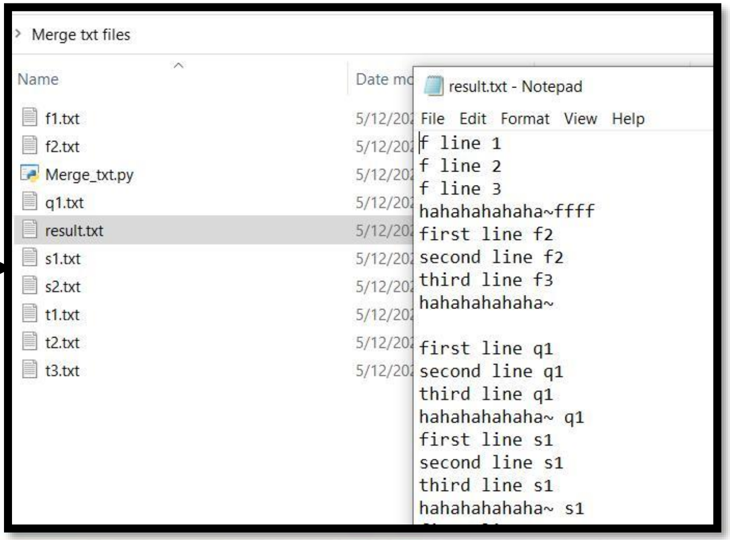
Practice 2023-09-22 Assignment I

- Write a python code to do:
  - Merge multiple text files in the current folder
  - Save as result.txt
  - Example



Name	Date modified
f1.txt	5/12/2021 8:50 PM
f2.txt	5/12/2021 8:50 PM
Merge_txt.py	
q1.txt	
s1.txt	
s2.txt	
t1.txt	
t2.txt	
t3.txt	

```
f line 1
f line 2
f line 3
hahahahahaha~ffff
```



Name	Date mo
f1.txt	5/12/202
f2.txt	5/12/202
Merge_txt.py	5/12/202
q1.txt	5/12/202
result.txt	5/12/202
s1.txt	5/12/202
s2.txt	5/12/202
t1.txt	5/12/202
t2.txt	5/12/202
t3.txt	5/12/202

```
f line 1
f line 2
f line 3
hahahahahaha~ffff
first line f2
second line f2
third line f3
hahahahahaha~

first line q1
second line q1
third line q1
hahahahahaha~ q1
first line s1
second line s1
third line s1
hahahahahaha~ s1
```

Practice 2023-09-22 Assignment II

- Write a python code to do:
  - In current folder, there exists multiple text files.
  - For each text file
    - check each line of the content,
      - If there exist the term ‘line’
      - Replace ‘line’ by ‘square’

Reminder, rules so far

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to improve your programming and problem solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly
6. If it was hard to write, it is probably hard to read. Add a comment.
7. All input is evil, unless proven otherwise.
8. A function should do one thing.