

Containers: Lists, Sets and Tuples

Introduction: Collections of values can be used for various types of analysis.

With a collection of numeric values, we can perform many types of analysis that would not be possible with a simple count or sum.

We will summarize a year's worth of data in the Fama-French file as we did previously, but be able to say much more about it.

```
var = [1, 4.3, 6.9, 11, 15]                                # a list container

print 'count is {}' .format(len(var))                      # count is 5
print 'sum is {}' .format(sum(var))                        # sum is XXXX
print 'average is {}' .format(sum(var) / len(var))         # average is XXXX

print 'max val is {}'.format(max(var))                    # max val is 15
print 'min val is {}'.format(min(var))                    # min val is 1

print 'top two: {}, {}'.format(var[3], var[4])            # top two: 11, 15

print 'median is {}' .format(var[len(var) / 2])          # median is XXXX
```

Introduction: Collections of values can be used to determine membership.

Checking one list against another is a core task in data analysis. We can validate arguments to a program, see if a user id is in a "whitelist" of valid users, see if a product is in inventory, etc.

We will apply membership testing to a *spell checker*, which simply checks every word in a file against a "whitelist" of correctly spelled words.

```
valid_actions = ['run', 'stop', 'search', 'reset']

input = raw_input('please enter an action: ')

if input in valid_actions:                                # if string can be found i
    print 'great, I will "{}".format(input)
else:
    print 'sorry, action not found'
```

Objectives for this Unit (Containers: Lists, Sets and Tuples)

Containers broaden our data analysis powers significantly over simple looping and summing. We can:

- Use lists to build up sequences of non-unique values.
- Use sets to build up collections of unique values.
- Use summary functions to summarize numeric data in containers (sum, max, min, etc.)
- Use sorting and slicing to do ordered analysis (top 5, median, etc.)
- Use membership analysis (**in**) to check a value against a collection of values.

Container Objects: List, Set, Tuple

Compare and contrast the characteristics of each container.

- **list**: ordered, *mutable* sequence of objects
- **tuple**: ordered, *immutable* sequence of objects
- **set**: unordered, mutable, unique collection of objects
- **dict**: unordered, mutable collection of object *key-value pairs*, with unique keys (discussed upcoming)

Summary for object: "List" container object

A **list** is an *ordered sequence* of values.

Initialize a List

```
var = []                                # initialize an empty list

var2 = [1, 2, 3, 'a', 'b']             # initialize a list of values
```

Append to a List

```
var = []

var.append(4)                          # Note well! call is not assigned
var.append(5.5)                        # list is changed in-place

print var                             # [4, 5.5]
```

Slice a List (compare to string slicing)

```
var2 = [1, 2, 3, 'a', 'b']             # initialize a list of values

sublist = var2[2:4]                    # [3, 'a']
```

Subscript a List

```
mylist = [1, 2, 3, 'a', 'b']           # initialize a list of values

xx = mylist[3]                         # 'a'
```

Get Length of a List (compare to **len()** of a string)

```
mylist = [1, 2, 3, 'a', 'b']

yy = len(mylist)                       # 5 (# of elements in mylist)
```

Test for membership in a List

```

mylist = [1, 2, 3, 'a', 'b']

if 'b' in mylist:                # this is True for mylist
    print "'b' can be found in mylist"  # this will be printed

print 'b' in mylist              # "True": the in operator actually
                                # returns True or False

```

Loop through a List (compare to looping through a file)

```

mylist = [1, 2, 3, 'a', 'b']

for var in mylist:
    print var                    # prints 1, then 2, then 3, then a, then b

```

Sort a List: **sorted()** returns a list of sorted values

```

mylist = [4, 9, 1.2, -5, 200, 20]

smyl = sorted(mylist)           # [-5, 1.2, 4, 9, 20, 200]

```

Summary for object: "Set" container object

A **set** is an *unordered, unique* collection of values.

Initialize a Set

```

myset = set()                   # initialize an empty set

mixed_set = set(['a', 9999, 4.3])  # the arg to set() is a list!

```

Add to a Set

```

myset = set()                                # initialize an empty set

mixed_set.add('a')                           # note well not assigned to a variable
mixed_set.add(4.3)

print mixed_set                              # set([4.3, 'a'])      (order is not

```

Get Length of a Set

```

mixed_set = set(['a', 9999, 4.3])

setlen = len(mixed_set)                      # 3

```

Test for membership in a Set

```

myset = set(['b', 'a', 'c'])
if 'c' in myset:                             # test is True
    print "'c' is in myset"                 # this will be printed

```

Loop through a Set

```

myset = set(['b', 'a', 'c'])
for el in myset:
    print el                                # will be printed in 'random' order

```

Sort a Set: **sorted()** returns a list of sorted object values

```

myset = set(['b', 'a', 'c'])

zz = sorted(myset)                          # ['a', 'b', 'c']

```

Summary for object: "Tuple" container object

A **tuple** is an *immutable ordered* sequence of values. Immutable means it cannot be changed once initialized.

Initialize a Tuple

```
var = ('a', 'b', 'c', 'd')    # initialize an empty tuple
```

Slice a Tuple

```
var = ('a', 'b', 'c', 'd')  
varslice = var[1:3]          # ('b', 'c')
```

Subscript a Tuple

```
mytup = ('a', 'b', 'c')  
last = mytup[2]              # 'c'
```

Get Length of a Tuple

```
mytup = ('a', 'b', 'c')  
tuplen = len(mytup)  
  
print tuplen                  # 3
```

Test for membership in a Tuple

```
mytup = ('a', 'b', 'c')  
if 'c' in mytup:  
    print "'c' is in mytup"
```

Loop through a Tuple

```
mytup = ('a', 'b', 'c')  
for el in mytup:  
    print el
```

Sort a Tuple

```
xxxx = ('see', 'i', 'you', 'ah')  
  
yyyy = sorted(xxxx)                # ('ah', 'i', 'see', 'you')
```

Summary for functions: len(), sum(), max(), min()

Summary functions offer a speedy answer to basic analysis questions: how many? How much? Highest value? Lowest value?

```
mylist = [1, 3, 5, 7, 9]           # initialize a list  
mytup = (99, 98, 95.3)             # initialize a tuple  
myset = set([2.8, 2.9, 1.7, 3.8]) # initialize a set  
  
print len(mylist)                  # 5  
print sum(mytup)                   # 292.3 sum of values in mytup  
print min(mylist)                  # 1 smallest value in mylist  
print max(myset)                   # 3.8 largest value in myset
```

Summary for function: sorted()

The **sorted()** function takes any sequence as argument and returns a list of the elements sorted by numeric or string value.

```
x = set([1.8, 0.9, 15.2, 3.5, 2])  
  
y = sorted(x)                      # [0.9, 1.8, 2, 3.5, 15.2]
```

Irregardless of the sequence passed to **sorted()**, a list is returned.

Summary task: Adding to Containers

We can add to a list with **append()** and to a set with **add**.

Add to a list

```
intlist1 = [1, 2, 55, 4, 9]           # list of integers

intlist1.append('hello')

print intlist1                        # [1, 2, 55, 4, 9, 'hello']
```

Add to a set

```
mixed_set = set(['a', 9999, 4.3])     # initialize a set with a list or

mixed_set.add('a')                    # not added – duplicate
mixed_set.add('cool')

print mixed_set                       # set(['a', 9999, 'cool'])
```

We cannot add to a tuple, of course, since they are immutable!

Summary task: Looping through Containers

We can loop through any container with **for**, just like a file.

Loop through a List

```
mylist = ['a', 'b', 'c']
for el in mylist:
    print el
```

Loop through a Set


```
myset = set(['b', 'a', 'c'])
for el in myset:
    print el                                # will be printed in 'random' c
```

Loop through a Tuple

```
mytup = ('a', 'b', 'c')
for el in mytup:
    print el
```

Loop through a String(??)

```
mystr = 'abcdefghi'

for x in mystr:
    print x                                # what do you see?
```

Summary task: Subscripting and Slicing Containers

We can slice any *ordered* container -- for us, list or tuple.

```
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
first_four = letters[0:4]
print first_four                          # ['a', 'b', 'c', 'd']

# no upper bound takes us to the end
print letters[5:]                         # ['f', 'g', 'h']
```

Remember the rules with slices:

- 1) the 1st index is 0
- 2) the lower bound is the 1st element to be included
- 3) the upper bound is one above the last element to be included
- 4) no upper bound means "to the end"; no lower bound means "from 0"

We cannot subscript or slice a set, of course, because it is unordered!

Summary task: Checking for Membership

We can check for value membership of a value within any container with **in**.

```
mylist = [1, 2, 3, 'a', 'b']

if 'b' in mylist:                    # this is True for mylist
    print "'b' can be found in mylist" # this will be printed
```

Summary task: Sorting Containers

Sorting allows us to rank values, find a median, and more.

```
mylist = [9.3, 2.1, 0.8]
xxx = sorted(mylist)                # a list: [0.8, 2.1, 9.3]

names = set(['David', 'George', 'Adam'])
yyy = sorted(names)                 # a list: ['Adam', 'David', 'George']

ints = (5, 9, 0, 8)
zzz = sorted(ints)                  # a list: [0, 5, 8, 9]
```

No matter what sequence is passed to **sorted()**, a list is returned. What about a string?!

Summary Exception: AttributeError

An **AttributeError** exception usually means calling a method on an object type that doesn't support that method.

Summary Exception: IndexError

An **IndexError** exception indicates use of an index for a list/tuple element that doesn't exist.

Practical: looping through a data source and building up containers

The "summary algorithm" is very similar to building a float sum from a file source. We loop; select; add.

list: build a list of states

```
state_list = []                                # initialize an empty list
for line in open('student_db.txt'):

    elements = line.split(':')
    state_list.append(elements[3])              # add the state for this row to st

chosen_state = raw_input('enter a state ID:  ')
state_freq = state_list.count(chosen_state)
print '{} occurs {} times'.format(chosen_state, state_freq)
```

set: build a set of unique states

```
state_set = set()                              # initialize an empty set
for line in open('student_db.txt'):

    elements = line.split(':')
    state_set.append(elements[3])              # add the state for this row to sta

chosen_state = raw_input('enter a state ID:  ')

if chosen_state in state_set:
    print 'that is a valid state'
else:
    print 'that is not a valid state'
```

Practical: checking for membership

We use **in** to compare two collections.

In this example, we have a **list** of ids and a **set** of valid ids. With looping and **in** we can build a list of valid and invalid ids.

```
student_states = ['CA', 'NJ', 'VT', 'ME', 'RI', 'CO', 'NY']
ne_states = set(['ME', 'VT', 'NH', 'MA', 'RI', 'CT'])

ne_student_states = []
for state in student_states:
    if state in ne_states:
        ne_student_states.append(state)

print 'students in our school are from these New England states: ', ne_st
```

This kind of analysis can also be done purely with **sets** and we'll discuss these methods later in the course.

Practical: treating a file as a list

Data files can be rendered as lists of lines, and slicing can manipulate them holistically rather than by using a counter.

In this example, we want to skip the 'header' line of the **student_db.txt** file. Rather than count the lines and skip line 1, we simply treat the entire file as a list and slice the list as desired:

```

fh = open('../python_data/student_db.txt')
file_lines_list = fh.readlines()      # a list of lines in the file
print file_lines_list
    # [ "id:address:city:state:zip",
    #   "jk43:23 Marfield Lane:Plainview:NY:10023",
    #   "ZXE99:315 W. 115th Street, Apt. 11B:New York:NY:10027",
    #   "jab44:23 Rivington Street, Apt. 3R:New York:NY:10002" ]

wanted_lines = file_lines_list[1:]    # take all but 1st element (i
for line in wanted_lines:
    print line.rstrip()               # jk43:23 Marfield Lane:Plain
                                     # ZXE99:315 W. 115th Street,
                                     # jab44:23 Rivington Street,

```

Sidebar: removing a container element

We rarely need to remove elements from a container, but here is how we do it.

```

mylist = ['a', 'hello', 5, 9]
myset = set([1, 3, 9, 11, 16])

popped = mylist.pop(0)  # remove the first element from mylist
                        # (argument specifies the index to remove)

mylist.remove(5)        # remove an element by value

myset.pop()             # remove a random element
myset.remove(3)         # remove an element by value

```