Lecture 6 Lists, Tuples and Dictionaries

IDS 400
Programming for Data Science in Business

- Assignment 3 is on blackboard
- Lab 5 is on blackboard

Quiz (1) Statistics

Minimum Value	48.00
Maximum Value	100.00
Range	52.00
Average	87.80
Median	92.00

Tentative schedule

Date	Lecture Number	Topics
08/24	Lecture 1	Introduction
08/31	Lecture 2	Basic
09/07	Lecture 3	Condition
09/14	Lecture 4	Loop
09/21	Lecture 5	String + **Quiz 1**
09/28	Lecture 6	Type
10/05	Lecture 7	Function
10/12	Lecture 8	File + **Quiz 2**
10/19	Lecture 9	Pandas
10/26	Lecture 10	Numpy
11/02	Lecture 11	Machine Learning
11/09	Lecture 12	Visualization
11/16	Lecture 13	Web Scraping & Deep Learning
11/23	***Thanksgiving***	***No lecture***
11/30	**Final presentation**	In class presentation
12/05	**Project submission due**	

For This Class

- Data Types
  - List
  - Tuple
  - Dictionary

List

List

- A *list* is a kind of collection.
- A collection allows us to put many values in a single “variable”

  - scores = [50, 60, 90]
  - friends = [‘alice’, ‘bob’, ‘charle’]
  - Empty_list_1 = list()
  - Empty_list_2 = [ ]

List constants

- List constants are surrounded by **square brackets** and the elements in the list are separated by **commas**.
- A list element can be *any Python object* – even another list.
- A list can be *empty*.

```
list_1 = [1, 23,45]
list_2 = ['red','blue']
list_3 = [25,'green']
list_4 = [1,[3,4],37]

print(list_1)
print(list_2)
print(list_3)
print(list_4)
```

```
[1, 23, 45]
['red', 'blue']
[25, 'green']
[1, [3, 4], 37]
```

Accessing elements

- We can get at any single element in a list using an index specified in square brackets.
- Index starts from **0**.
- Index must be **integer**.
- If the index is a *negative* value, it counts *backward from the end of the list*.

```
x = [1,3,45,10]
print(x[0])
print(x[1])
print(x[-1])
print(x[-3])
```

```
1
3
10
3
```

Elements	**1**	**3**	**45**	**10**
Index (+)	0	1	2	3
Index (-)	-4	-3	-2	-1

List length

- The *len()* function takes a list as a parameter and returns the number of elements in the list.

```
numbers1 = [1, 3, 45, 10]
print(len(numbers1))
```

```
4
```

```
numbers2 = [1, [3, 45], 10]
print(len(numbers2))
```

```
3
```

Lists are mutable

- Lists are "**mutable**" –we can change an element of a list using index operator.

Elements	**1**	**3**	**45**	**10**
Index (+)	0	1	2	3

Elements	**1**	**3**	**33**	**10**
Index (+)	0	1	2	3

```
x = [1,3,45,10]
x[2] = 33
print(x)
```

```
[1, 3, 33, 10]
```

The range function

- The *range(**m**)* function returns a list of numbers that range from zero to m-1.
- The *range(**x, y**)* function returns a list of numbers that range from x to y-1.
- The *range(**x, y, z**)* function returns a list of numbers that range from x to y-1 with a step z.
- If x > y, returns an empty list.

```
for i in range(4):
    print(i)
```

```
0
1
2
3
```

```
for i in range(3,9):
    print(i)
```

```
3
4
5
6
7
8
```

```
for i in range(1,9,2):
    print(i)
```

```
1
3
5
7
```

```
for i in range(4,1):
    print(i)
```

List membership

- *in* is a ***boolean*** operator that tests membership in a sequence.
- *not in* is to test whether an element is NOT a member of a list.
- They do NOT modify the list.

```
fruit = ['apple','banana','orange']
print('banana' in fruit)
```

```
True
```

```
x = [3,4,5,6,7,8]
print(7 not in x)
```

```
False
```

List and loops

- The generalized syntax of a *for* loop with lists is:

```
for variable in ListName:
    <statement>
    <statement>
    …
<statement>
```

Iterate a list using ***while*** loop

```
i = 0
while i < len(ListName):
    variable = ListName[i]
     <statement>
     i = i + 1
```

- Here, variable takes the value of the item inside the list on each iteration.
- Loop continues until we reach the last item in the list.

List and loops

- *for* loop

  i represents elements

```
x = range(3,6)
sum1 = 0.0
for i in x:
    sum1 += i
avg = sum1/len(x)
print("average is: ", avg)
```

```
average is:  4.0
```

- *while* loop

i represents the index of elements

```
x = range(3,6)
sum2 = 0.0
i = 0
while i<len(x):
    sum2 += x[i]
    i = i+1
avg = sum2/len(x)
print("average is: ", avg)
```

```
average is:  4.0
```

List operations

- We can create a new list by adding two existing lists together.

```
a = [1,2,3]
b = [4,5,6]
c = a + b
print(c)
```

```
[1, 2, 3, 4, 5, 6]
```

```
d = ['x','y']
e = a + d
print(e)
```

```
[1, 2, 3, 'x', 'y']
```

List operations

- Lists can be sliced using colon *:*
  - ListName[**x** : **y**] returns a sublist from index **x** to index **y – 1**.
  - ListName[: **y**] returns a sublist from index **0** to index **y – 1**.
  - ListName[**x** :] returns a sublist from index **x** to the **end**.

```
a = [9,41,12,3,77,19]
print(a[1:3])
```

```
[41, 12]
```

```
print(a[:4])
```

```
[9, 41, 12, 3]
```

```
print(a[3:])
```

```
[3, 77, 19]
```

```
print(a[1:4:2])
```

```
[41, 3]
```

Repeated steps

- Using slices *:* to delete elements is awkward.
- Python provides an alternative that more readable ***del***.
  - ***del listName***[**i**] delete the element with index **i**.
  - ***del listName***[**i** : **j**] delete the element with index from **i** to **j - 1**.

```
a = [9,41,12,3,77,19]
a[1:3] = []
print(a)
```

```
[9, 3, 77, 19]
```

```
del a[1]
print(a)
```

```
[9, 77, 19]
```

```
del a[:2]
print(a)
```

```
[19]
```

List method - append

- Building a list from scratch
  - We can create an empty list and then add elements using the ***append*** method.
  - The list *stays in order* and new elements are added *at the end of the list*.

```
a = list()
print(a)
```

```
[]
```

```
a.append('book')
a.append(30)
print(a)
```

```
['book', 30]
```

List method - sort

- A list is an ordered sequence.
  - A list can be sorted (i.e., change its order)
  - The ***sort*** method sorts the list ascending by default. If *reverse* is *True*, the list is sorted in descending order.

```
a = ['Joseph','Glenn','Sally']
a.sort()
print(a)
```

```
['Glenn', 'Joseph', 'Sally']
```

```
a = ['Joseph','Glenn','Sally']
a.sort(reverse=True)
print(a)
```

```
['Sally', 'Joseph', 'Glenn']
```

List method - reverse

- *reverse()* does not return any value but reverse the given object from the list.

```
a = ['Joseph','Glenn','Sally']
a.reverse()
print(a)
```

```
['Sally', 'Glenn', 'Joseph']
```

Some built-in functions

```
a = [3,44,13,11,77,15]
print(len(a))
```

```
6
```

```
print(max(a))
```

```
77
```

```
print(min(a))
```

```
3
```

```
print(sum(a))
```

```
163
```

```
print(sum(a)/len(a))
```

```
27.166666666666668
```

Using External Library

```
from statistics import mean

mean(a)
```

```
27.166666666666668
```

An exercise

- Request several input from users, then calculate the average of user inputted numbers.

Example

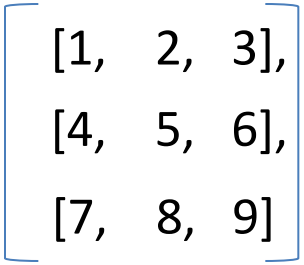
- Request several input from users, and then calculate the average of user inputted numbers.

```
numList = list()
while True:
    inp1= input('Enter a number: ')
    if inp1== 'done':
        break
    value = float(inp1)
    numList.append(value)
average = sum(numList) / len(numList)
print('Average: ',average)
```

```
Enter a number: 1
Enter a number: 2
Enter a number: 3
Enter a number: 7
Enter a number: 8
Enter a number: 9
Enter a number: done
Average:  5.0
```

Matrices

- Nested lists are often used to represent matrices.




```
matrix = [[1,2,3],[4,5,6],[7,8,9]]
print(matrix[1])
```

```
[4, 5, 6]
```

```
print(matrix[1][2])
```

```
6
```

Tuples

Mutability

- A **tuple** is similar to a list except that it is **immutable**. The elements of a tuple can NOT be modified.
- A tuple is a comma-separated list of values. Parenthesis is not necessary but recommended.

```
tuple1 = 'a','b','c'
print(tuple1)
```

```
('a', 'b', 'c')
```

```
tuple2 = ('a','b',1)
print(tuple2)
```

```
('a', 'b', 1)
```

```
tuple2[2] = 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-36-4d434f6e5c67> in <module>
----> 1 tuple2[2] = 5

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

Things NOT to do with tuples

```
x = (3, 2, 1)
x.sort()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-37-90ddb0001caf> in <module>
      1 x = (3, 2, 1)
----> 2 x.sort()

AttributeError: 'tuple' object has no attribute 'sort'
```

```
x.append(4)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-38-61ee22000a19> in <module>
----> 1 x.append(4)

AttributeError: 'tuple' object has no attribute 'append'
```

```
x.reverse()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-44-1cc9031a0932> in <module>
----> 1 x.reverse()

AttributeError: 'tuple' object has no attribute 'reverse'
```

Tuples are more efficient

- Since Python does not have to build tuple structures to be modifiable, they are simpler and more efficient in terms of memory use and performance than lists.
- So, in our program when we are making “temporary variables”, we prefer tuple over lists.

Tuple

- To create a tuple with a single element, we have to include the final comma.

```
tuple1 = ('a',)
```

- All slice operation are similar to lists.
- Though we can not modify the elements of a tuple, we can replace it with a different tuple.

```
tuple1 = ('a','b','c')
tuple1 = ('A',) + tuple1[1:]
print(tuple1)
```

```
('A', 'b', 'c')
```

Tuples and assignment

- We can put a tuple on the *left hand* side of an assignment statement.
- We can even omit the parenthesis.
- To swap two values, we can use tuple assignment to neatly solve this problem.

```
(x, y) = (4,'hello')
print(y)
```

```
hello
```

```
a, b = (1,7)
a, b = b, a
print(a)
print(b)
```

```
7
1
```

List/Tuples are comparable

- The comparison operators work with tuples and other sequences. If the first item is equal, Python goes on to the next element, and so on, until it finds elements that differ.

```
(0, 1, 2) < (0, 1, 3)
```

```
True
```

```
(0, 1, 2000) < (0, 2, 2)
```

```
True
```

```
('Joseph','Glenn','Sally') > ('Glenn','Sally')
```

```
True
```

Random numbers

- The *random* function in random module returns a floating point number between 0.0 and 1.0.
- The *randint(start, end)* function returns an integer from start to end.

```
import random
x = random.random()
print(x)
```

```
0.6090977442877039
```

```
y = random.randint(1,8)
print(y)
```

```
6
```

Dictionaries

Dictionaries

- The compound types we have learned: lists and tuples – use integers as indices.
- **Dictionaries** are similar to these type except that they can use other types as an index.
- Create an empty dictionary:

```
eng2sp = {}
eng2sp = dict()
```

Dictionaries

- Dictionaries are like lists except that they use ***keys*** instead of numbers to look up values.

```
eng2sp = {}
eng2sp['one'] = 'uno'
eng2sp['two'] = 'dos'
print(eng2sp)
```

```
{'one': 'uno', 'two': 'dos'}
```

```
ddd = dict()
ddd['age'] = 21
ddd['score'] = 90
print(ddd)
```

```
{'age': 21, 'score': 90}
```

```
ddd['age'] = 23
print(ddd)
```

```
{'age': 23, 'score': 90}
```

Dintionaries

- Dictionaries are like bags – ***no order***.

```
purse = {}
purse['money'] = 12
purse['candy'] = 3
purse['tissues'] = 75
print(purse)
```

```
{'money': 12, 'candy': 3, 'tissues': 75}
```

```
print(purse['candy'])
```

```
3
```

```
purse['candy'] = purse['candy'] + 2
print(purse)
```

```
{'money': 12, 'candy': 5, 'tissues': 75}
```

Dictionary operations

- *del* statement removes key-value pair from a dictionary.
- We can also change the value associated with a key.
- It is an error to reference a key which is not in the dictionary.

```
ddd = dict()
ddd['age'] = 21
ddd['score'] = 90
print(ddd)
```

```
{'age': 21, 'score': 90}
```

```
del ddd['age']
print(ddd)
```

```
{'score': 90}
```

```
print(ddd['course'])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-68-8408d53237b8> in <module>
----> 1 print(ddd['course'])

KeyError: 'course'
```

Dictionary methods

- *dictName.keys()* returns a list of the keys in the dictionary.
- *dictName.values()* returns a list of the values in the dictionary.
- *dictName.items()* returns both, in the form of a list tuples – one for each key-value pair.

```
ddd = {'age':21,'score':90}
ddd.keys()
```

```
dict_keys(['age', 'score'])
```

```
ddd.values()
```

```
dict_values([21, 90])
```

```
ddd.items()
```

```
dict_items([('age', 21), ('score', 90)])
```

Dictionary methods

- We can use the *in* operator to see if a key is in the dictionary.
- In Python 2, dictName.has_key() returns True if the key appears in the dictionary. However, this method is no longer available in Python 3.

```
ddd = {'age':21,'score':90}
ddd.keys()
```

```
dict_keys(['age', 'score'])
```

```
'age' in ddd
```

```
True
```

Dictionary methods

- ***dictName.get(key)*** returns the value if the key appears in the dictionary.
- ***dictName.get(key, 0)*** returns the value if the key appears in the dictionary, 0 otherwise. Advantage of this function is it will not throw an error message.

Exercise

Count the frequency of elements in a list and save the result in a dictionary.

Solution

- We can use *if* and *else* statement.

```
counts = dict()
names = ['csev','cwen','csev','zqian','cwen']
for name in names:
    if name not in counts:
        counts[name]=1
    else:
        counts[name] = counts[name] + 1
print(counts)
```

```
{'csev': 2, 'cwen': 2, 'zqian': 1}
```

Dictionary methods

- We can solve this problem easily using ***dictName.get(key, 0)*** method.

```
counts = dict()
names = ['csev','cwen','csev','zqian','cwen']
for name in names:
    counts[name] = counts.get(name,0) + 1
print(counts)
```

```
{'csev': 2, 'cwen': 2, 'zqian': 1}
```

Iterate dictionaries

- Iterating through **Keys** directly.
- Iterating through **.Keys()**.
- Iterating through **.values()**.
- Iterating through **.items()**.

```
a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
for i in a_dict:
    print(i)
```

```
color
fruit
pet
```

```
for i in a_dict.keys():
    print(i)
```

```
color
fruit
pet
```

```
for i in a_dict.values():
    print(i)
```

```
blue
apple
dog
```

```
for i in a_dict.items():
    print(i)
```

```
('color', 'blue')
('fruit', 'apple')
('pet', 'dog')
```

Iterate dictionaries

- Two iteration variables
  - We loop through the key-value pairs in a dictionary using two iteration variables.
  - Each iteration, the first variable is the key and the second variable is the corresponding value for the key.

```
statesAndCapitals = {
                     'Gujarat' : 'Gandhinagar',
                     'Maharashtra' : 'Mumbai',
                     'Rajasthan' : 'Jaipur',
                     'Bihar' : 'Patna'
                    }

print('List Of given states and their capitals:\n')

# Iterating over values
for state, capital in statesAndCapitals.items():
    print(state, ":", capital)
```

```
List Of given states and their capitals:

Gujarat : Gandhinagar
Maharashtra : Mumbai
Rajasthan : Jaipur
Bihar : Patna
```

Duplicate elements

```
dup_list = ['a','b','c','a',1,1]
print(dup_list)
```

```
['a', 'b', 'c', 'a', 1, 1]
```

```
dup_tuple = ('a','b','c','a',1,1)
print(dup_tuple)
```

```
('a', 'b', 'c', 'a', 1, 1)
```

- In lists and tuples, duplicate elements are supported.

Duplicate elements

- Dictionaries do not support duplicate keys. However, we can have duplicate values in a dictionary.

```
# dictionary - duplicate key
dup_dict = {'a':1,'b':2,'c':3,'a':4}
print(dup_dict)
```

```
{'a': 4, 'b': 2, 'c': 3}
```

```
# dictionary - duplicate value
dup_dict = {'a':1,'b':2,'c':1}
print(dup_dict)
```

```
{'a': 1, 'b': 2, 'c': 1}
```

Containers as dictionary values

- More than one value can correspond to a single key using a list.

For example, with the dictionary {"a": [1, 2]}, 1 and 2 are both connected to the key "a"

```
# list as values
a_dictionary = {"a": [1, 2], "b": [3, 4]}
print(a_dictionary)
```

```
{'a': [1, 2], 'b': [3, 4]}
```

```
# tuple as values
a_dictionary = {"a": (1, 2), "b": (3, 4)}
print(a_dictionary)
```

```
{'a': (1, 2), 'b': (3, 4)}
```

```
# dictionary as values
a_dictionary = {"a": {"a":1, "b":2}, "b": {"a":1, "b":2}}
print(a_dictionary)
```

```
{'a': {'a': 1, 'b': 2}, 'b': {'a': 1, 'b': 2}}
```

How to access the value of a “history” subject

```
sampleDict = {
   "class":{
      "student":{
         "name":"Mike",
         "marks":{
            "physics":70,
            "history":80
         }
      }
   }
}
```

How to access the value of a “history” subject

```
sampleDict = {
   "class":{
      "student":{
         "name":"Mike",
         "marks":{
            "physics":70,
            "history":80
         }
      }
   }
}
```

```
sampleDict["class"]["student"]["marks"]["history"]
```

```
80
```

Another data type - Set

A set is a collection which is both **unordered** and **unindexed**.

- Sets are written with curly brackets.
- Every set element is **unique** (no duplicates) and must be **immutable** (cannot be changed).
- However, a set itself is mutable. We can add or remove items from it.

```
# set of integers
my_set = {1, 2, 3}
print(my_set)

# set of mixed datatypes
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)
```

```
{1, 2, 3}
{1.0, (1, 2, 3), 'Hello'}
```

```
# we can make set from a list
my_set = set([1, 2, 3, 2])
print(my_set)
```

```
{1, 2, 3}
```

```
# set cannot have duplicates
my_set = {1, 2, 3, 4, 3, 2}
print(my_set)
```

```
{1, 2, 3, 4}
```

```
# set cannot have mutable items
# here [3, 4] is a mutable list
# this will cause an error.
my_set = {1, 2, [3, 4]}
```

```
---------------------------------------------
TypeError                                 T
<ipython-input-11-bc1d6bd215a8> in <module>
      2 # here [3, 4] is a mutable list
      3 # this will cause an error.
----> 4 my_set = {1, 2, [3, 4]}

TypeError: unhashable type: 'list'
```

Data types

```
a = [1,2,3]
type(a)
```

```
b = (1,2,3)
type(b)
```

```
c = {1,2,3}
type(c)
```

```
d = {'height':1,'width':2}
type(d)
```

Data types

```
a = [1,2,3]
type(a)
```

```
list
```

```
b = (1,2,3)
type(b)
```

```
tuple
```

```
c = {1,2,3}
type(c)
```

```
set
```

```
d = {'height':1,'width':2}
type(d)
```

```
dict
```

Difference?

- Brackets
- Mutable
- Index
- Duplicates

Lab *Types*