UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

Essential Data Skills for Business Analytics

Lecture 5: Lists, Tuples, and Dictionaries

Decision, Operations & Information Technologies
Robert H. Smith School of Business
Spring, 2020

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']

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

```
>>> print([1,23,45])
[1,23,45]
>>> print(['red','blue'])
['red','blue']
>>> print([25,'green'])
[25,'green']
>>> print([1,[3,4],37])
[1,[3,4],37]
>>> print ([])
[]
```

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

1	3	45	10
0	1	2	3

Index:

```
>>> x = [1,3,45,10]

>>> print (x[1])

3
```

Accessing elements

- If the index is negative value, it counts backward from the end of the list

1	3	45	10
-4	-3	-2	-1

Index:

```
>>> x = [1,3,45,10]

>>> print (x[-1])

10
```

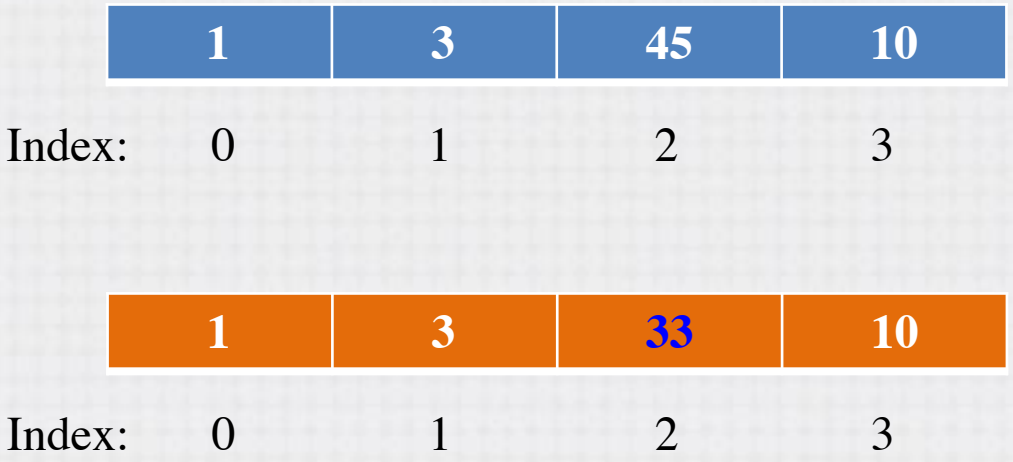
List length

- The len() function takes a list as a parameter and returns the number of elements in the list
- numbers1 = [1, 3, 45, 10]
- numbers2 = [1, [3, 45], 10]

```
>>> 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




```
>>> 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
- If x>y, returns an empty range

```
>>> print (list(range(4)))
[0,1,2,3]

>>> print (list(range(3,9)))
[3,4,5,6,7,8]

>>> print (list(range(4,1)))
[]
```

List membership

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

```
>>> fruit = ['apple','banana','orange']
>>> 'banana' in fruit
True

>>> x = [3,4,5,6,7,8]
>>> 2 not in x
True
```

Lists and for loops

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

for variable *in* ListName:
    Statements

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

```
x = range(3,6)
sum = 0.0
for i in x:
    sum += i
avg = sum/len(x)

print ("average is: ", avg)
```

```
x = range(3,6)
sum = 0.0
i = 0
While i<len(x):
    sum += x[i]
    i = i+1
avg = sum/len(x)

print ("average is: ", avg)
```

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 :
  - ListName[x:y] returns a sublist from index x to index y-1
  - ListName[:x] returns a sublist from index 0 to index x-1
  - ListName[x:] returns a sublist from index x to the end

```
>>> a = [9,41,12,3,77,19]
>>> a[1:3]
[41,12]

>>> a[:4]
[9,41,12,3]

>>> a[3:]
[3,77,19]
```

List operations

- Using slices : to delete list elements is error prone
- Python provides an alternative that more readable
  - del listName[i] delete the element with index i
  - del listName[i:j] delete elements 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 methods (1)

- 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() # a = []
>>> print (a)
[]
>>> a.append('book')
>>> a.append(30)
>>> print (a)
['book',30]
```

List methods (2)

- A list is an ordered sequence
  - A list can be sorted (i.e., change its order)
  - The sort method means "sort yourself"

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

>>> print (a[1])
Joseph
```

Built-in functions

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

```
>>> 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
```

Example


```
numList = list()

while True:
    inputs = input('Enter a number: ')

    if inputs == 'done':
        break

    value = float(inputs)

    numList.append(value)


average = sum(numList) / len(numList)

print ('Average: ',average)
```

Matrices

- Nested lists are often used to represent matrices.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
>>> matrix = [[1,2,3],[4,5,6],[7,8,9]]
>>> matrix[1]
[4,5,6]

>>> 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'

>>> tuple2 = ('a','b',1)
>>> tuple2[2] = 5  (ERROR!!!)
```

Things not to do with tuples

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

```
>>> x = (3, 2, 1)
>>> x.sort()
Traceback:AttributeError: 'tuple' object has no
attribute 'sort'
>>> x.append(5)
Traceback:AttributeError: 'tuple' object has no
attribute 'append'
>>> x.reverse()
Traceback: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
  - >>> a_tuple = ('a' ,)
- All slice operation are similar to lists
- Even we can not modify the elements of a tuple, we can replace it with a different tuple
  - >>> a_tuple = ('a', 'b', 'c')
  - >>> a_tuple = ('A' ,) + a_tuple[1:]
  - >>> print (a_tuple)
  - ('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)
7
```

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) < (5, 1, 2)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
>>> ( 'Jones', 'Sally' ) < ('Jones', 'Sam')
True
>>> ( 'Jones', 'Sally') > ('Adams', 'Sam')
True
```

Random numbers

- The random module contains a function called *random* that returns a **floating point number** between 0.0 and 1.0.

```
>>> import random
>>> x = random.random()
>>> print (x)
0.15156642489

>>> y = random.random()
>>> print (y)
0.32856673042
```

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 any immutable type as an index.
- Create an empty dictionary
  - eng2sp = {}
  - eng2sp = dict()

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

Dictionaries

- Dictionaries are like bags – no order

```
>>> purse = {}
>>> purse['money'] = 12
>>> purse['candy'] = 3
>>> purse['tissues'] = 75
>>> print (purse)
{'money':12, 'tissue':75, 'candy':3}
>>> print (purse['candy'])
3
>>> >>> purse['candy']=purse['candy']+2
>>> print (purse)
{'money':12, 'tissue':75, 'candy':5}
```

Lists vs. Dictionaries

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

```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(180)
>>> print (lst)
[21,180]

>>> lst[0] = 23
>>> print (lst)
[23,180]
```

```
>>> ddd = dict()
>>> ddd['age'] = 21
>>> ddd['score'] = 90
>>> print (ddd)
{'score':90, 'age':21}

>>> ddd['age'] = 23
>>> print (ddd)
{'score':90, 'age':23}
```

Dictionary operations

- del statement removes a 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['height'])
KeyError: 'height'
```

Dictionary methods

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

- dictName.keys() returns a list of the keys that appear
- 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()
['age','score']

>>> ddd.values()
[21,90]

>>> ddd.items()
[('age',21),('score',90)]
```

Dictionary methods

- dictName.has_key() returns true if the key appears in the dictionary
- We can also use the **in** operator to see if a key is in the dictionary

```
>>> ddd = {'age':21,'score':90}
>>> ddd.has_key('age')
True

>>> ddd.has_key('height')
False

>>> '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

```
counts = dict()

names = ['csev','cwen','csev','zqian','cwen']

for name in names:
    counts[name] = counts.get(name,0) + 1

print (counts)
```

Dictionary example

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

```
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)
```

Iterate dictionaries

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

```
>>> counts = {'chuck' : 1, 'fred' : 42, 'jan' : 100}
>>> for key in counts
...         print key,  counts[key]
...
jan 100
chuck 1
fred 42
```

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

```
students = {'name':'alice', 'age':20, 'gender': 'f'}

for k,v in students:
    print (k,":",v)

Outputs:
name: alice
age: 20
gender: f
```