PythonLectureDataStructures

January 30, 2018

0.1 Python Data Structures and Sequences

0.1.1 Tuple

A tuple is a fixed-length, immutable sequence of Python objects which means you cannot update or change the values of tuple elements

To create a tuple,

1. we use comma separated sequenc of values.
2. When you're defining tuples in more complicated expressions, it's often necessary to enclose the values in parentheses.
3. You can convert any sequence or iterator to a tuple by invoking tuple:

```
In [10]: mytup = 2017, 2018, 2019
         print(mytup)

         mytup2 = (2017, 2018, 2019)
         print(mytup2)
         #nested tuple
         tupnested = ((2017, 2018, 2019), ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday
         print(tupnested)
         mylist = [2017, 2018]
         print(type(mylist))
         tup1 = tuple(mylist)
         print(type(tup1))
         print(tup1)

         tup2 = tuple("January")
         print(tup2)
```

```
(2017, 2018, 2019)
(2017, 2018, 2019)
((2017, 2018, 2019), ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday'))
<class 'list'>
<class 'tuple'>
(2017, 2018)
('J', 'a', 'n', 'u', 'a', 'r', 'y')
```

0.1.2 Accessing Values in Tuples

Elements can be accessed with square brackets [] as with most other sequence types. As in C, C++, Java, and many other languages, sequences are **0-indexed in Python**:

```
In [14]: tup1[0]
         #tup1[4]
         tup1[0]= 3
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-14-59bcaa80981e> in <module>()
  1 tup1[0]
  2 #tup1[4]
----> 3 tup1[0]= 3


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

While the objects stored in a tuple may be mutable themselves, **once the tuple is created it's not possible to modify which object is stored in each slot**

If an object inside a tuple is mutable, such as a list, you can modify it in-place:

```
In [23]: t4 = tuple([2017, "String", True, [3.1415, 2.718]])
         t4[3].append(1)
         print(t4)
         #t4[2] = False
```

```
(2017, 'String', True, [3.1415, 2.718, 1])
```

0.1.3 Basic Tuples Operations

1. • operator: concatenate tuples using the + operator to produce longer tuples
2. • operatorMultiplying a tuple by an integer, as with lists, has the effect of concatenating together that many copies of the tuple
3. in operator: test the membership of the given object
4. len(): return the length of the tuple

```
In [27]: t1 = (2017, 2019)
         t2 = ("M", "T", "W", "R", "F")
         t3 = t1 + t2
         print(t3)
```

```
t4 = t3*3
t4

"M" in t3
print(len(t4))
```

```
(2017, 2019, 'M', 'T', 'W', 'R', 'F')
21
```

0.1.4 Unpacking tuples

If you try to assign to a tuple-like expression of variables, Python will attempt to unpack the value on the righthand side of the equals sign:

```
In [33]: t1 = (2017, 2018, 2019)
         a,b,c = t1
         b
         #a,b = t1
         #b
         t2 = (2017, 2018 , ("M", "T"))
         a,b, (c, d) = t2
         d
         a, b, c = t2
         c
         a, b = t2
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-33-cdf306e86ffb> in <module>()
      9 a, b, c = t2
     10 c
---> 11 a, b = t2


ValueError: too many values to unpack (expected 2)
```

typical Unpacking in Pthon

1. Swap variable in Python
2. iterating over sequence of tuples or lists
3. returning multiple values from a function

4. *rest: pluck" a few elements from the beginning of a tuple. This rest bit is sometimes something you want to discard; there is nothing special about the rest name. As a matter of convention, many Python programmers will use the underscore (_) for unwanted variables

```
In [40]: a= 1
         b =2
         tmp =a
         a = b
         b= tmp
         b
         #Pthon way
         a,b = b, a

         myseq = [ (1,2, 3), (4, 5, 6), (7, 8, 9)]
         for a, b, c in myseq:
             print("a = {0}, b= {1}, c = {2}".format(a,b,c))

         t5 = (1,2,3,4, 5,6,7)
         a,b, *rest  = t5
         print("a= {0}, b= {1}".format(a,b))

         a,b, *_  = t5
         print("a= {0}, b= {1}".format(a,b))
```

```
a = 1, b= 2, c = 3
a = 4, b= 5, c = 6
a = 7, b= 8, c = 9
a= 1, b= 2
a= 1, b= 2
```

0.2 List

In contrast with tuples, lists are variable-length and their contents can be modified in-place.
To create a list, you may use the following method

1.

2. list function:

```
In [51]: #empty list
         el = []
         len(el)

         #single type
         l1 = [3.14, 2.718, 1, 0]

         #list with mixed datatypes
         l3 = [ 2018, "Jan", 3.14, None]
```

```
len(l3)
#nested lists
l4 = [3.14, ["M", "F"], [2018, 2019]]
t1 = (3.14, ["M", "F"])
l5 =list(t1)
l5[1] = ["M", "T"]
l5
range(10)
```

```
Out[51]: range(0, 10)
```

0.3 Adding and removing elements

1. append function: append elemnts to the end of the list
2. insert: insertan element at a specific location in the list. The insertion index must be between 0 and the length of the list, inclusive.
3. pop: **removes and returns** an element at a given index
4. remove: remove the element by value, which locates the first such value and removes it from the list

```
In [56]: l1 = [2018, 2019]
         l1.append(2020)

         l1.insert(0,2017)
         print(l1)

         l1.pop(1)

         print(l1)
         l1. remove(2020)
         l1.remove(2000)
```

```
[2017, 2018, 2019, 2020]
[2017, 2019, 2020]
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-56-9219a498b8ce> in <module>()
  9 print(l1)
 10 l1. remove(2020)
---> 11 l1.remove(2000)
```

```
ValueError: list.remove(x): x not in list
```

0.3.1 List operators

1. in /not in : check whether the list contains a value
2. +: concatenate the lists and get a longer list
3. extend: append multiple elements to a list
4. sorting: sort a list in-place without creatng a new object

```
In [ ]: l10 = [1, 2, 3, ["M", "F"], 3.14]
        1 in l10
        "M" not in l10
        ["M", "F"] in l10

        l11 = [2017, 2018] + ["Jan", "Feb"]

        l11.extend(["T", "R"])
        print(l11)
```

Note that list concatenation by addition is a comparatively expensive operation since a new list must be created and the objects copied over. Using extend to append elements to an existing list, especially if you are building up a large list, is usually preferable. Thus,

```
In [ ]: import timeit
        start = timeit.default_timer()
        num = []
        mynum = (range(1000000))
        for i in mynum:
            num.extend([i**3])

        stop = timeit.default_timer()
        print("It takes: {0} seconds!".format(stop - start))
        #much fater than
        start = timeit.default_timer()
        num = []
        num = []
        mynum = (range(1000000))
        for i in mynum:
            num = num + [i**3]
        stop = timeit.default_timer()
        print("It takes: {0} seconds!".format(stop - start))
```

```
In [1]: l12 = [3, 5, 1, 10]
        l12.sort()
```

0.4 Index and Slicing

1. the index could be positive starting from **0** instead of 1; for exaple 0, 1, 2, ..., n-1
2. the index could be negative: for example for the last one to the first one indexed by -1, -2, ..., -n
3. object[start slice: end slice:step] which **include** the start slice but **exlcude** the end slice
4. missing slice indication: from start/up to end

```
In [14]: myl = [10, 20, 30, 40, 50]
         print(myl[0])
         print(myl[1])
         print(myl[2])

         print(myl[-1])
         print(myl[-2])

         print(myl[0:2])
         print(myl[:4])
         print(myl[3:])
         print(myl[:])

         print(myl[1:4:2])
         print(myl[-3:-1])
         print(myl[1:-1])
         print(myl[::-2])
         print(myl[:-1])
```

```
10
20
30
50
40
[10, 20]
[10, 20, 30, 40]
[40, 50]
[10, 20, 30, 40, 50]
[20, 40]
[30, 40]
[20, 30, 40]
[50, 30, 10]
[10, 20, 30, 40]
```

0.5 Built-in Sequence Functions

1. enumerate It's common when iterating over a sequence to want to keep track of the index of the current item and the value of the current term

2. sorted: returns a new sorted list form the elements of any sequence
3. zip "pairs" up the elements of a number of lists, tuples, or other sequences to create a list of tuples:
4. reversed iterates over the elements of a sequence in reverse order

```
In [17]: #typical approach in other language
         i = 0
         for value in collection:
         # do something with value
             i += 1


        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-17-c4f836e21604> in <module>()
          1 #typical approach in other language
          2 i = 0
    ----> 3 for value in collection:
          4 # do something with value
          5     i += 1


        NameError: name 'collection' is not defined


In [18]: #Python has a built-in function, enumerate, which returns a
         #sequence of (i, value) tuples:
         for i, value in enumerate(collection):
         # do something with value


          File "<ipython-input-18-50c246d1d06a>", line 4
        # do something with value
                                   ^
    SyntaxError: unexpected EOF while parsing



In [26]: some_list = ['foo', 'bar', 'baz']
         mapping = {}
         for i, v in enumerate(some_list):
             mapping[v] = i
         print(mapping)

         sorted([7, 1, 2, 6, 0, 3, 2])
```

```
seq1 = ['foo', 'bar', 'baz']
seq2 = ['one', 'two', 'three']
zipped = zip(seq1, seq2)
list(zipped)

seq3 = [False, True]
list(zip(seq1, seq2, seq3))
for i, (a, b) in enumerate(zip(seq1, seq2)):
    print('{0}: {1}, {2}'.format(i, a, b))
```

```
{'foo': 0, 'bar': 1, 'baz': 2}
0: foo, one
1: bar, two
2: baz, three
```

Given a "zipped" sequence, zip can be applied in a clever way to "unzip" the sequence. Another way to think about this is converting a list of rows into a list of columns. The syntax, which looks a bit magical, is:

```
In [30]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),
          ('Schilling', 'Curt')]
         first_names, last_names = zip(*pitchers)
         print(first_names)
```

```
('Nolan', 'Roger', 'Schilling')
```

```
In [31]: list(reversed(range(10)))
```

```
Out[31]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

0.6 Dictionary dic {"key":"value"}

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

```
In [36]: empth_dic = {}

         d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}
         #You can access, insert, or set elements using the same syntax as for accessing element
         #of a list or tuple, we need to use the key instead index
         d1[7] = "an integer"
         d1
         d1['b']
```

```
        #You can check if a dict contains a key using the same syntax used for checking
        #whether a list or tuple contains a value:
        'a' in d1
```

```
Out[36]: True
```

You can delete values either using the del keyword or the pop method (which simultaneously returns the value and deletes the key):

```
In [39]: d1[5] = "some value"
         print(d1)
         del d1[5]
         print(d1)
         ret = d1.pop('a')
         print(ret)
         print(d1)
```

```
{'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer', 5: 'some value'}
{'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
some value
{'b': [1, 2, 3, 4], 7: 'an integer'}
```

The keys and values method give you iterators of the dict's keys and values, respectively. While the key-value pairs are not in any particular order, these functions output the keys and values in the same order:

```
In [43]: list(d1.keys())

         list(d1.values())
```

```
Out[43]: [[1, 2, 3, 4], 'an integer']
```

You can merge one dict into another using the update method:

```
In [45]: d1.update({'b' : 'foo', 'c' : 12})
         print(d1)
```

```
{'b': 'foo', 7: 'an integer', 'c': 12}
```

0.6.1 Creating dicts from sequences

It's common to occasionally end up with two sequences that you want to pair up element-wise in a dict. As a first cut, you might write code like this:

```
In [46]: mapping = {}
         for key, value in zip(key_list, value_list):
             mapping[key] = value
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-46-011c26510e49> in <module>()
  1 mapping = {}
----> 2 for key, value in zip(key_list, value_list):
  3     mapping[key] = value


NameError: name 'key_list' is not defined
```

Since a dict is essentially a collection of 2-tuples, the dict function accepts a list of 2-tuples:

```
In [48]: mapping = dict(zip(range(5), reversed(range(5))))
         print(mapping)
```

```
{0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

0.7 Valid dict key types

While the values of a dict can be any Python object, the keys generally have to be immutable objects like scalar types (int, float, string) or tuples (all the objects in the tuple need to be immutable, too). The technical term here is hashability. You can check whether an object is hashable (can be used as a key in a dict) with the hash function:

```
In [49]: hash('string')
         hash((1, 2, (2, 3)))
         hash((1, 2, [2, 3])) # fails because lists are mutable
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-49-83cee15ba612> in <module>()
  1 hash('string')
  2 hash((1, 2, (2, 3)))
----> 3 hash((1, 2, [2, 3])) # fails because lists are mutable


TypeError: unhashable type: 'list'
```

To use a list as a key, one option is to convert it to a tuple, which can be hashed as long as its elements also can:

```
In [50]: d = {}
         d[tuple([1, 2, 3])] = 5
```

0.8 Set

A set is an unordered collection of unique elements. You can think of them like dicts, but keys only, no values. A set can be created in two ways:

1. via the set function
2. via a set literal with curly braces:

The elements in a set are **unique**, there is no duplicated in a set.

```
In [51]: set([2, 2, 2, 1, 3, 3])
         {2, 2, 2, 1, 3, 3}

Out[51]: {1, 2, 3}
```

0.8.1 Sets Operations

1. union: The union of these two sets is the set of distinct elements occurring in either set. This can be computed with either the **union method or the | binary operator**
2. intersection : The intersection contains the elements occurring in both sets. The **& operator or the intersection method** can be used:
3. difference
4. symmetric difference

```
In [53]: a = {1, 2, 3, 4, 5}
         b = {3, 4, 5, 6, 7, 8}
         a.union(b)
         a|b

         a.intersection(b)
         a&b

Out[53]: {3, 4, 5}
```

0.9 List, Set, and Dict Comprehensions

List comprehensions are one of the most-loved Python language features. They allow you to concisely form a new list by filtering the elements of a collection, transforming the elements passing the filter in one concise expression.

```
In [56]: #A common programming task is to iterate over a list and transform each element
         squares = []
         for num in range(10):
             squares.append(num**2)
         squares
         #list comprehensions allow the same idea to be expressed in much fewer lines.
         squares = [x**2 for x in range(10)]
         squares
```

```
Out[56]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The basic syntax for list comprehensions is this: [expr for val in collection if condition]

```
In [57]: #This is equivalent to the following for loop:
         result = []
         for val in collection:
             if condition:
                 result.append(expr)


         ---------------------------------------------------------------------------

         NameError                                 Traceback (most recent call last)

         <ipython-input-57-c1705c6714a8> in <module>()
           1 #This is equivalent to the following for loop:
           2 result = []
         ----> 3 for val in collection:
           4     if condition:
           5         result.append(expr)


         NameError: name 'collection' is not defined


In [58]: [x for x in range(10) if x % 2 == 0]

Out[58]: [0, 2, 4, 6, 8]

In [59]: #The following list comprehension generates the squares of even numbers and the cubes o
         # Python

         [x**2 if x % 2 == 0 else x**3 for x in range(10)]

Out[59]: [0, 1, 4, 27, 16, 125, 36, 343, 64, 729]

In [2]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
        [x.upper() for x in strings if len(x) > 2]

Out[2]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Set and dict comprehensions are a natural extension, producing sets and dicts in an idiomatically similar way instead of lists. A dict comprehension looks like this:

dict_comp = {key-expr : value-expr for value in collection if condition}

A set comprehension looks like the equivalent list comprehension except with curly braces instead of square brackets: set_comp = {expr for value in collection if condition}

```
In [5]: unique_lengths = {len(x) for x in strings}
        unique_lengths
        set(map(len, strings))

        loc_mapping = {val : index for index, val in enumerate(strings)}
```

0.10 Nested list comprehensions

```
In [7]: a = [[1,2],[3,4],[5,6]]
        b = [x for xs in a for x in xs]
        b

Out[7]: [1, 2, 3, 4, 5, 6]

In [8]: # It is similar to the following nested loops
        b = []
        for xs in a:
           for x in xs:
               b.append(x)
```

```
  +----------------a-----------------+
  | +--xs---+ , +--xs---+ , +--xs---+ | for xs in a
  | | x , x |   | x , x |   | x , x | | for x in xs
```

a = [[1 , 2] , [3 , 4] , [5 , 6]] b = [x for xs in a for x in xs] == [1,2,3,4,5,6] #a list of just the "x"s

```
In [9]: all_data = [['John', 'Emily', 'Michael', 'Mary', 'Steven'],
            ['Maria', 'Juan', 'Javier', 'Natalia', 'Pilar']]
```

Now, suppose we wanted to get a single list containing all names with two or more e's in them. We could certainly do this with a simple for loop:

```
In [11]: names_of_interest = []
         for names in all_data:
             enough_es = [name for name in names if name.count('e') >= 2]
             names_of_interest.extend(enough_es)
         print(names_of_interest)

['Steven']


In [13]: result = [name for names in all_data for name in names
             if name.count('e') >= 2]
         print(result)

['Steven']


In [14]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
         flattened = [x for tup in some_tuples for x in tup]
         print(flattened)

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

In [15]:
```
# Keep in mind that the order of the for expressions would be the same if you wrote a
# nested for loop instead of a list comprehension
flattened = []
for tup in some_tuples:
    for x in tup:
        flattened.append(x)
```

In [17]:
```
#This produces a list of lists, rather than a flattened list of all of the inner elemen
a = [[x for x in tup] for tup in some_tuples]
print(a)
```

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