Week 7 Agenda

Week 7 Tuples and Dictionaries

Chapter 6 and 16 in Python Crash Course: A Hands-on, Project-Based Introduction to Programming

Tuple

A **tuple** is part of the standard language. This is a data structure very similar to the **list** data structure. The main difference being that tuple manipulation are faster than list because tuples are immutable which means that it can not be modified.

Like a list, a tuple is typically created in an assignment statement:

<tupleVariable> = (<element1>, <element2>, ... <elementN>)

An example of creating a Tuple of friends is shown below:

```
>>> friendsTuple = ('Joe', 'Martha', 'John', 'Susan')
>>> print(friendsTuple)
('Joe', 'Martha', 'John', 'Susan')
>>>
```

Dictionary

A dictionary is similar to a list in that it allows you to refer to a collection of data by a single variable name. However, it differs from a list in one fundamental way. In a list, order is important and the order of the elements in a list never changes (unless you explicitly do so). Because the order of elements in a list is important, you refer to each element in a list using its index (its position within the list).

In a dictionary, the data is represented in what are called ***key/value*** pairs. The keys within a dictionary must be unique. The syntax of a dictionary looks like this:

```
{<key>:<value>, <key>:<value>, ..., <key>:<value>}
```

Dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place

Dictionary Usage

```
#Create a new dictionary called “telephone”
>>> telephone = {'mike': '402-555-1212', 'barb': '314-231-8973', 'andy': '515-643-9087'}

#adds a new element to the dictionary
>>> telephone['kelly'] = '402-330-9870’

#Print the dictionary
>>> telephone
{'kelly': '402-330-9870', 'mike': '402-555-1212', 'barb': '314-231-8973', 'andy': '515-643-9087'}
>>> print(telephone)
{'kelly': '402-330-9870', 'mike': '402-555-1212', 'barb': '314-231-8973', 'andy': '515-643-9087'}

#Access the value of the "mike" key’
>>> telephone['mike']
402-555-1212’

#Assign a new value for an existing key in a dictionary
>>> telephone[’mike'] = ’402-396-9078’ #change value of an existing key
```

Find Dictionary Keys and Values

There are two additional operations (functions) that you can use on a dictionary. If you want, you can find all the keys defined in a dictionary with a call to <dictionary>.keys(). You can find all the values with a call to <dictionary>.values(). Both calls return a pseudo-list. Here is an example using our previously defined dictionary:

```
>>> print(telephone.keys())
dict_keys(['kelly', 'mike', 'barb', 'andy'])

 >>> print(telephone.values())
dict_values(['402-330-9870', '402-555-1212', '314-231-8973', '515-643-9087'])
```

Neither dict_keys nor dict_values are subscriptable like a Python list, meaning you can't access a specific item using dict_keys[2]. In order to use them as a list you would use:

```
>>> keys = list(telephone.keys())
>>> print keys[2]
'mike'
```

In Keyword

To ensure that we are using a valid key, we can use the in operator before attempting to use a key in a dictionary. The in operator is used like this:

```
<key> in <dictionary>
```

Example:

```
’mike’ in telephone
```

You can also use a similar syntax to see if a value is in the keys or the values of a dictionary.

```
>>> 'mike' in telephone.keys()
True
>>> 'mike' in telephone.values()
False
>>>
```

Checking Values with “If” Statements

In any code where we think that a key might not be found, it's a good idea to add some defensive coding to check and ensure that the key is in the dictionary before we attempt to use it on the dictionary. Typically, we build this type of check using an if statement:

```
if myKey in myDict:
    # OK, we can now successfully use myDict[myKey]
else:
    # The key was not found, print some error message or take some other action
```

Sometimes, it may make more logical sense to code the reverse test. We can use not in to test for the key not being in the dictionary:

```
if myKey not in myDict:
    # The key was not found, do whatever you need to do
```

Iterate Through a Dictionary

There are several ways to iterate through a dictionary using a for loop. In the example below we use the for loop to iterate through the keys in the dictionary.

```
>>> statesDict = {\ 'California':38802000, 'Texas':26956000, 'Florida':19893000,
'New York':19746000,\ 'Illinois': 12880000, 'Pennsylvania': 12787000,
'Ohio':11594000, 'Georgia': 10097000,\ 'North Carolina': 9943964,
'Michigan':9909000, 'New Jersey': 8938000}

>>> for state in statesDict:
          population = statesDict[state]
          print state, population
```

Iterate through Dictionaries using items()

When looping through dictionaries, the key and corresponding value can also be retrieved at the same time using the items() method. Python automatically assigns the first variable as the name of a key in that dictionary, and the second variable as the corresponding value for that key.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for key, value in knights.items():
    print("{}: {}".format(key, val))

gallahad: the pure
Robin: the brave
```

List of Dictionaries

It is common to combine lists and dictionaries. In the example below we have a list of cars called carsList. Each car dictionary has a make, model, year, doors, and mileage key with an associated value. Below is an example of the list as well as an iteration example. The overall list is defined by the [ ] and each dictionary within the list is defined by the { }

```
carsList = [{'make':'Toyota', 'model':'Prius', 'year': 2006, 'doors':4, 'mileage': 65000},
            {'make':'Honda', 'model':'Civic', 'year': 2010, 'doors':2, 'mileage': 54321},
            {'make':'Ford', 'model':'Fusion', 'year': 2012, 'doors':4, 'mileage': 24680},
            {'make':'Chevy', 'model':'Volt', 'year': 2015, 'doors':4, 'mileage': 7890}]

for carDict in carsList:
    if (carDict['doors'] == 4) and (carDict['mileage'] < 50000):
        print(carDict['make'], carDict['model'], carDict['license'])
```

JSON Data

JSON (JavaScript Object Notation) is a common format of data that is often used by applications to send data to other applications. It is common for an application that uses a 3rd party API to receive data back from that system in JSON format. The interesting thing about JSON data is that it resembles Python Dictionaries which makes it extremely easy to work with.

JSON Example Program

This program will request data from the YAHOO Weather API. The data is provided back in JSON format.

```
import urllib2, urllib, json #import the libraries we'll need
from pprint import pprint #import pretty print

baseurl = "https://query.yahooapis.com/v1/public/yql?"

yql_query = "select item.condition from weather.forecast where woeid in (select woeid from
geo.places(1) where text='omaha, ne')"
yql_url = baseurl + urllib.urlencode({'q':yql_query}) + "&format=json"

result = urllib2.urlopen(yql_url).read() #make an http request to Yahoo
data = json.loads(result) #loads the JSON data into a Python object
print(type(data)) #print the object type of our data variable that holds our result
pprint(data) #pprint does a "pretty print" which makes the data more readable
```

JSON Example Program Results

When you run this program you notice the results that are printed are in JSON format which resembles a Python Dictionary.

```
D:\tools>python weather.py
{u'query': {u'count': 1,
            u'created': u'2018-02-06T17:06:33Z',
            u'lang': u'en-US',
            u'results': {u'channel': {u'item': {u'condition': {u'code': u'26',
                                                               u'date': u'Tue, 06 Feb 2018 10:00 AM CST',
                                                               u'temp': u'6',
                                                               u'text': u'Cloudy'}}}}}
```

References

The Basics of Compiled Languages, Interpreted Languages, and Just-in-Time Compiliers, https://www.upwork.com/hiring/development/the-basics-of-compiled-languages-interpreted-languages-and-just-in-time-compilers/

The End

You may close this Window and return to the course.