Python for Informatics

LESSON 5

Dictionaries

- A ***dictionary*** is similar to a ***list***.
- The difference has to do with how you select an element.
- With a ***list***, you select an element by means of an index position, or in other words an integer.
- With a dictionary, you select an element by means of a ***key***.

Dictionaries

- A ***list*** is a sequence of elements that can be accessed by integer indexes.
- A ***dictionary*** is not a sequence.
- A ***dictionary*** does not have an order to it.
- ***A dictionary is a mapping of keys to values.***
- A ***dictionary*** is a ***set of key-value pairs***.

Dictionaries

- While the keys and values can be of nearly any type, ***keys are most frequently of type string***.

```
eng2sp = dict()
eng2sp['funny'] = 'chistoso'
eng2sp['sad'] = 'triste'
print(eng2sp)
{'funny': 'chistoso', 'sad': 'triste'}
```

Dictionaries

- Again, there is no regular order to a dictionary.

    eng2sp['happy'] = 'allegre'

    eng2sp['relaxed'] = 'tranquilo'

    print(eng2sp)

    {'funny': 'chistoso', 'relaxed': 'tranquilo', 'happy': 'allegre', 'sad': 'triste'}

Dictionaries

- Again, there is no regular order to a dictionary.

```
eng2sp['happy'] = 'allegre'
eng2sp['relaxed'] = 'tranquilo'
print(eng2sp)
{'funny': 'chistoso', 'relaxed': 'tranquilo', 'happy': 'allegre', 'sad': 'triste'}
```

Dictionaries

- Order has no bearing upon the use of a dictionary.
- You look up any given element value based upon its associated key.

  print(eng2sp['happy'])

  allegre

  print(eng2sp['relaxed'])

  tranquilo

- A ***dictionary*** is an ***associative lookup***.

Dictionaries

- The ***len()*** function returns the number of key-value pairs of a given dictionary.

  print(len(eng2sp))

  4

- The ***in*** operator indicates whether or not an operand is a key within a given dictionary.

  'funny' in eng2sp

  True

Dictionaries

- Note that the ***in*** operator searches for a key, not a value.

'chistoso' in eng2sp

False

'relaxed' in eng2sp

True

Dictionaries

- The ***values()*** function gets the ***list*** of ***values*** in a dictionary.

```
vals = eng2sp.values()
print(vals)
['chistoso', 'tranquilo', 'allegre', 'triste']
```

Dictionaries

- Therefore, if you want to know if a specified ***value*** exists within a ***dictionary***, you can combine the ***in*** operator with the ***values()*** function.

```
vals = eng2sp.values()
'chistoso' in vals
True
...or...
'chistoso' in eng2sp.values()
True
```

Dictionaries

- ***lists*** use a linear search algorithm.
- As the size of a ***list*** grows, the average search time increases linearly.
- Python ***dictionaries*** are ***implemented as hash tables***, which has the effect reducing search time to be about the same no matter how large the ***dictionary*** grows.
- **Large dictionaries are fast; large lists,… not so much.**

Dictionaries

- Let us consider a use case wherein we employ a ***dictionary*** as a set of counters.
- For a given ***string***, you need to count the number of times each character appears in the ***string***.
- While there are many approaches to this problem, we will consider three.

Dictionaries

1. Create 26 ***variables***, where each one is a counter for the number of times that a given character appears as you traverse the ***string***. With this approach you would likely use a chained (if…elif…) conditional.
2. Create a ***list*** of 26 ***elements***. Given a character ***value***, convert it to an ***integer*** (using the ***ord()*** function, and some normalizing math).

Dictionaries

3. Create a ***dictionary***, where the characters are your ***keys***, and the ***values*** are counters of the number of times that each character appears as you traverse the ***string***. The first time you see a character, you add it to the ***dictionary*** with a value of 1. Thereafter, when you see a character again you increase its corresponding counter value.

Dictionaries

- Each one of the foregoing approaches will work.
- One advantage to the ***dictionary*** approach is that you only need to create elements for the characters that actually appear in the string—you don't need to create 26 elements!

Dictionaries

- Here's the code:

```
word = 'onomatopoeia'
d = dict()
for c in word:
    if c not in d:
        d[c] = 1
    else:
        d[c] += 1
print d
{'a': 2, 'e': 1, 'i': 1, 'm': 1, 'o': 4, 'n': 1, 'p': 1, 't': 1}
```

Dictionaries

- Our resulting ***dictionary*** is a ***histogram***.
- Each ***dictionary*** item is a mapping of a character ***key*** to the frequency of occurance ***value*** of that character.

Dictionaries

- The ***dictionary get()*** method accepts a ***key*** and a default ***value*** as arguments.
- The ***get()*** method searches for the ***key***, and if it finds it, returns the ***value*** associated with that ***key***. If the method doesn't find the ***key***, then it returns the given default ***value***.

Dictionaries

- Here's an example of the ***get()*** method in action:

```
names = {'wynken': 1, 'blynken': 5, 'nod': 42}
print(names.get('nod', 0))
42
print(names.get('tim', 0))
0
```

Dictionaries

- Because the ***get()*** method takes care of the case where the ***key*** is not found, we can employ it to make our histogram code more concise:

```
word = 'onomatopoeia'
d = dict()
for c in word:
    d[c] = d.get(c, 0) + 1
print d
{'a': 2, 'e': 1, 'i': 1, 'm': 1, 'o': 4, 'n': 1, 'p': 1, 't': 1}
```

Dictionaries

- Now that we understand how to use ***dictionaries***, and we've already learned how to use ***files***, let's combine them together!
- Our code will read each line of text from the ***file***, parse each line into a ***list*** of words, and then create a ***dictionary*** histogram of the words.

Dictionaries


```
fname = raw_input('Please enter the file name:')
Please enter the file name:C:\UCSD\PythonForInformatics\code\romeo.txt
try:
   fhand = open(fname)
except:
    print('Cannot open file: ', fname)
    exit()
    word_counts = dict()
   for line in fhand:
       words = line.split()
       for word in words:
           if word not in word_counts:
               word_counts[word] = 1
           else:
               word_counts[word] += 1
 print word_counts
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1, 'is': 3, 'through': 1, 'pale': 1,
'yonder': 1, 'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1, 'window': 1, 'sick': 1,
'east': 1, 'breaks': 1, 'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1, 'kill': 1, 'the': 3,
'soft': 1, 'Juliet': 1}
```

Dictionaries

- Although our output is correct and complete, its format is not easily readable by the human eye.
- We can write some code to make our presentation more appealing.
- To keep things simple, consider this:

```
names = {'wynken': 1, 'blynken': 5, 'nod': 42}
for key in names:
    print key, names[key]
blynken 5
nod 42
wynken 1
```

Dictionaries

- Going back to our Shakespeare example, upon closer inspection we find that we have a couple annoying problems.
- The ***split()*** function operates with spaces as a default delimiter, which means that the words ***soft!*** and ***soft*** will be seen as different words, and hence will be given different counts.
- Similarly, the words ***Who*** and ***who*** will be seen as different, and will be processed separately.

Dictionaries

- To solve these two problems, we can use the string constant ***punctuation***, and the string methods ***lower()*** and ***translate()***.
- ***punctuation*** is a string constant that specifies all of the characters that are considered to be punctuation characters.
- The ***lower()*** method returns a copy of the string with all characters converted to lower case.

Dictionaries

- The ***translate()*** method is a bit more complicated:

  string.translate(s, table[, deletechars])

- ***translate()*** returns a string that has been created such that any and all characters in ***s*** have been removed, and the remaining characters are translated using ***table*** (being a 256 character string giving the translation by ordinal indexing). If table is ***None***, then the translation is not performed.
- We don't need ***table***, but we do want to remove punctuation characters.

Dictionaries

- Finally, here is out new and improved code:

```
import string
fname = raw_input('Please enter the file name:')
Please enter the file name:C:\UCSD\PythonForInformatics\code\romeo.txt
try:
   fhand = open(fname)
except:
   print('Cannot open file: ', fname)
   exit()
   word_counts = dict()
   for line in fhand:
      line = line.translate(None, string.punctuation)        # These two lines
      line = line.lower()                                     # are new.
      words = line.split()
      for word in words:
         if word not in word_counts:
            word_counts[word] = 1
         else:
            word_counts[word] += 1
print word_counts
```

Tuples

- A ***tuple*** is very much like a ***list***.
- Like a ***list***, a ***tuple*** is a sequence of values.
- Unlike a ***dictionary***, a ***tuple*** does **not** store ***key/value pairs***.

Tuples

- Like other ***sequences***, a ***tuple*** is ***comparable***—you can easily compare one ***tuple*** to another.
- ***tuples*** are also ***hashable***—they can be sorted, and they can conveniently serve as ***keys*** in ***dictionaries***.

Tuples

- The big difference from a ***list*** is that… a ***tuple*** is ***immutable***—you cannot change a ***tuple***!
- This ***immutability*** is what makes ***tuples hashable***, and hence usable as ***keys***.

Tuples

- The simplest syntax for creating a ***tuple*** is to specify a series of comma separated values:

t = 'zero', 'one', 'two', 'three', 'four', 'five'
print(t)
('zero', 'one', 'two', 'three', 'four', 'five')

Tuples

- For clarity, it is a best practice to enclose ***tuple*** definitions with parenthesis—this helps to quickly identify them as ***tuples*** as opposed to ***lists***.

  t = ('zero', 'one', 'two', 'three', 'four', 'five')
  print(t)
  ('zero', 'one', 'two', 'three', 'four', 'five')

- While ***lists*** use brackets, ***[]***, ***tuples*** use parentheses ***()***.

Tuples

- To create a ***tuple*** with a single item, you must be sure to include the ***comma***.

```
tup1 = ('one',)
type(tup1)
tuple
```

Tuples

- Watch what happens when we forget to include the all-important ***comma***.

  tup2 = ('one')

  type(tup2)

  str

- Without the ***comma***, the python interpreter assumes we are providing a parenthesized expression that evaluates to a ***str***.

Tuples

- The built-in function ***tuple()*** gives us yet another way to create a ***tuple***:

    quest = tuple('Holy Grail')

    print(quest)

    ('H', 'o', 'l', 'y', ' ', 'G', 'r', 'a', 'i', 'l')

- Notice how the ***tuple()*** function takes a ***sequence*** and creates a ***tuple*** from the elements of the given ***sequence***.

Tuples

- If you do not pass in an argument to the ***tuple()*** function, an ***empty tuple*** is returned.

zilch= tuple()

print(zilch)

()

Tuples

- The ***tuple()*** function is an example of a ***constructor function***.
- ***Constructors*** construct or create ***object instances***, and the ***tuple()*** function constructs ***tuples***.
- Given that ***tuple()*** is the name of a constructor, you should avoid using ***tuple*** as the name of a variable.

Tuples

- Many of the ***list*** operators also work with ***tuples***.
- Bracket operator:

  quest = ('H', 'o', 'l', 'y', ' ', 'G', 'r', 'a', 'i', 'l')

  print(quest[0])

  H
- Slice operator:

  print(quest[5:10])

  ('G', 'r', 'a', 'i', 'l')

Tuples

- Remember, however, that an attempt to modify a ***tuple*** is a big **no no!**

quest[0] = 'P'

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-40d5d9eb7ce0> in <module>()
----> 1 quest[0] = 'P'

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

Tuples

- If you don't like your current tuple, then just associate the name to a different tuple (assign a different tuple to your variable).

quest = ('H', 'o', 'l', 'y', ' ', 'G', 'r', 'a', 'i', 'l')
quest = ('P', 'o', 'l', 'y', ' ', 'G', 'r', 'a', 'i', 'l')
print(quest)
('P', 'o', 'l', 'y', ' ', 'G', 'r', 'a', 'i', 'l')

Tuples

- The comparison operators work with ***tuples***, ***lists***, and ***sequences*** in general.
- ***sequences*** are compared by comparing each respective element of the two ***sequences***, beginning with the first element, and then each successive element in turn.
- Short-circuit evaluation is observed, meaning that as soon a difference is identified the comparison is evaluated and the result is affirmed.

Tuples

 # tuple comparison – one element at a time

(0, 1, 2, 3) < (0, 1, 2, 4)

True

 # Short-circuit evaluation 2 < 3 is returned

(0, 1, 2, 3000) < (0, 1, 3, 4)

True

Tuples

- The ***sort()*** method works similarly, by comparing elements 0, then 1, then 2, only as needed to sort the ***tuples*** out.

```
tup_lst = [(3, 2, 4), (3, 1, 5)]
tup_lst.sort()
print(tup_lst)
[(3, 1, 5), (3, 2, 4)]
```

- In the above example, elements 0 and 1 are compared, but elements 2 (values 4 and 5) are not, because the sort order has already been determined.

Tuples

- This short-circuit element comparison feature of ***tuples*** enables a pattern called **DSU**, which stands for **Decorate, Sort, and Undecorate**.
- The context of this **DSU** pattern involves the use of a ***list*** of ***tuples***, such that the ***tuples*** are sorted by means of the ***list sort()*** method.

Tuples

- **Decorate** – We "decorate" by building a ***list*** of ***tuples*** such that they contain one or more ***sort keys*** (keys that serve as a basis for sorting).
- **Sort** – We "sort" by invoking the ***sort()*** method of our ***list*** of decorated ***tuples***.
- **Undecorate** – We "undecorate" by extracting the ***value elements*** of our sorted ***tuples***.

Tuples


```
phrase = 'only the finest baby frogs'
words = phrase.split()
tup_lst = list()
for word in words:
    tup_lst.append((len(word), word))
tup_lst.sort(reverse = True)
des_len_lst = list()
for length, word in tup_lst:
    des_len_lst.append(word)
print(des_len_lst)
['finest', 'frogs', 'only', 'baby', 'the']
```

Tuples

- ***Tuples*** can be placed on the left side of an assignment operator.

```
lst = ['baby', 'frogs']
w1, w2 = lst
print(w1)
print(w2)
baby
frogs
```

- Note that when ***tuples*** are initialized in this way on the left side of an assignment operator, the parenthesis are usually omitted.
- Also, note that whereas the ***tuple*** elements are named (***w1*** and ***w2***), the ***tuple*** itself is not.

Tuples

- ***Tuple*** assignment syntax also facilitates the swapping of our ***named tuple elements*.**

```
w1, w2 = w2, w1
print(w1)
print(w2)
frogs
baby
```

Tuples

- ***Tuple*** assignment syntax can be generalized to accommodate any kind of ***sequence*** (***string***, ***list***, or ***tuple***) on the right side of the assignment operator.

```
addr = 'monty@python.org'
uname, domain = addr.split('@')
print(uname)
print(domain)
monty
python.org
```

Dictionaries & Tuples

- ***Dictionaries*** have an ***items()*** method that returns a ***list*** of its ***tuples***, where each ***tuple*** is a ***key-value pair***.

```
dict = {'blynken': 50, 'nod': 75, 'wynken': 25}
tup_lst = dict.items()
print(tup_lst)
[('blynken', 50), ('nod', 75), ('wynken', 25)]
tup_lst.sort()
print(tup_lst)
[('blynken', 50), ('nod', 75), ('wynken', 25)]
```

Dictionaries & Tuples

- Now let's combine ***items()***, ***tuple assignment***, and ***for*** to traverse the values of a ***dictionary*** in a loop:

for key, val in dict.items():
    print val, key

- Note that the above loop has two ***iteration variables***.
- ***items()*** returns a ***list*** of ***tuples***, and each ***tuple*** is assigned per iteration as ***key-value pairs***.
- The key order is determined by the ***dictionary*** hashing algorithm.

Dictionaries & Tuples


```
dict = {'blynken': 50, 'nod': 75, 'wynken': 25}
lst = list()
for key, val in dict.items():
    lst.append( (val, key) )    # Neat trick!

print(lst)
lst.sort(reverse = True)
print(lst)
[(50, 'blynken'), (75, 'nod'), (25, 'wynken')]
[(75, 'nod'), (50, 'blynken'), (25, 'wynken')]
```

- Our ***list*** is constructed such that the ***dictionary's values*** are the ***list's keys***, and vice versa.
- This gives us a ***list*** allowing for the ***sorting*** of our ***dictionary's values***.

- We can use the ***dictionary to list of sorted values trick*** to print the ten most common words in the romeo-full.txt file.

  # See next slide for code.

Dictionaries & Tuples


```
import string
fhand = open('C:/UCSD/PythonForInformatics/code/romeo-full.txt')
counts = dict()
for line in fhand:
    line = line.translate(None, string.punctuation)
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1
# Sort the dictionary by value
lst = list()
for key, val in counts.items():
    lst.append( (val, key) )

lst.sort(reverse = True)

for key, val in lst[:10] :
    print key,val
```

Dictionaries & Tuples

61 i
42 and
40 romeo
34 to
34 the
32 thou
32 juliet
30 that
29 my
24 thee

Dictionaries & Tuples

- **The foregoing demonstration of Python's information parsing and analyzing prowess goes a long way toward explaining why Python is such a popular language for data analytics!**

Dictionaries & Tuples

- Remember, ***tuples*** are ***hashable***; ***lists*** are not.
- If we need a ***composite key*** *(a key composed of two or more values)* for a ***dictionary***, we must use a ***tuple*** as our ***key***.

Dictionaries & Tuples

- We can build a phone directory by using successive tuple assignments, as with the statement…

  num_direct[last, first, birthdate] = number

- Once the dictionary is populated with key-value pairs, we traverse and process it as follows:

  for last, first, birthdate in num_direct:

  print first, last, birthdate, num_direct[last, first, birthdate]

- Note that the expression in brackets is a tuple that serves as an index key.

Dictionaries & Tuples

- We have looked specifically at ***lists*** of ***tuples***.
- Additional possibilities include ***lists*** of ***lists***, ***tuples*** of ***tuples***, ***tuples*** of ***lists***, etc.
- Suffice it to say that there are many possible permutations of nested structures.
- Structures can also go deeper, as in ***lists*** of ***lists*** of ***lists***…

Dictionaries & Tuples

- ***lists*** are used more frequently than ***tuples***, because they are ***mutable***.
- For return statements, ***tuples*** are often syntactically simpler to create than a ***list***.
- ***tuples*** make good ***dictionary keys***.
- Arguments to functions/methods are often best passed as ***tuples***, as they avoid ***aliasing*** ambiguities.