LING115 Lecture note
Session #5: Python dictionaries

1. Introduction

So far we have learned the following data types: Boolean values, integers, floating points, strings, and lists. We will learn another data type called dictionaries.

2. Dictionaries

A dictionary in Python is a set of key-value pairs. We index individual values in a dictionary by their keys. A key and its value are paired with a colon, and the set of pairs is enclosed in braces. See the following example:

```python
>>> a = { 'x': 'a', 'y': 'b' }
>>> a['x']
'a'
>>> a['y']
'b'
```

We must index a key that exists in the dictionary. Otherwise, we get a `KeyError` as in the following example:

```python
>>> a['z']
```

A key must be of an immutable type: a list cannot be a key, for example. For example, the following returns a `TypeError`.

```python
>>> a = { [1,2,3]:1 }
```

A data type is mutable if you can change its items individually by assignment operator. Lists are of mutable sequence type in Python. Consider the following example:

```python
>>> a = [ 'x','y','z' ]
>>> a[1] = 'w'
>>> a
['x','w','z']
```

On the other hand, strings are not mutable types. Try the following examples and you will get `TypeError`.

```python
>>> b = 'xyz'
>>> b[1] = 'w'
```
Anyway, going back to dictionaries, a dictionary key must be of immutable data-type so lists cannot be used as a key. If you want to use a list as a key, you must first convert it to an immutable type. One way to do this is to convert the list to a corresponding tuple, which is a data-type very similar to a list (see section 5.3 at [http://docs.python.org/tutorial/datastructures.html](http://docs.python.org/tutorial/datastructures.html)). A list can be converted to a tuple and vice versa as in the following example:

```python
>>> a = [1,2,3]
>>> b = tuple(a)
>>> c = list(b)
```

Once a list is converted to a tuple, it can be used as a dictionary key.

```python
>>> a = [1,2,3]
>>> b = tuple(a)
>>> d = {b:1}
```

We can add a new key-value pair to a dictionary as follows:

```python
>>> a = {}
>>> a[1] = 'a'
```

In the first line above, we created an empty dictionary. In the next line, we added a new key-value pair, where 1 is the key and ‘a’ is the value.

We can remove a key-value pair from a dictionary by using del as follows:

```python
>>> del a[1]
```

Below are some useful methods for dictionary, where `d` is a dictionary object.

`d.keys()`

Returns the list of keys in the dictionary.

`d.has_key(x)`

Returns whether the dictionary has a key `x`.

`d.items()`

Returns a list of key-value pairs in the dictionary.

`d.values()`

Returns a list of values in the dictionary.

`d.get(x,y)`

Returns `d[x]`, if the dictionary has a key `x`. Otherwise, it returns `y`. 
3. **Exercise: counting words**

We can write a Python program that counts words from a corpus using a dictionary. This is not that different from how we created a list of unique words from a corpus: instead of updating a list of words, we update a dictionary of word-frequency pairs. Here is what we can do:

- Read lines from standard input into a list called `lines`.
- **Create an empty dictionary called** `freq_dict`.
- For each line in `lines`, do the following:
  - Remove any leading or trailing white space characters.
  - Split line by white space and store it as a list of words called `word_list`.
  - For each word in `word_list`, do the following:
    - Convert word into lower-case.
    - If word is already a key of `freq_dict`, add one to its existing value.
    - If not, add word as a key of `freq_dict` and 1 as its value.
- For each key of `freq_dict`, do the following:
  - Print out key and its value.

We already know how to implement all this in Python except the ones in boldface.

An empty dictionary called `freq_dict` can be created as follows:

```python
freq_dict={}
```

We can use `has_key()` to check if a word is already used as a key of the dictionary or not. So the two conditional statements above can be implemented as follows:

```python
if freq_dict.has_key(word):
    freq_dict[word]=freq_dict[word]+1
else:
    freq_dict[word]=1
```

Finally, we can use `keys()` to list all the keys of the dictionary. So one way to print out the key-value pairs would look something like the following:

```python
for word in freq_dict.keys():
    print word+\t+str(freq_dict[word])
```

Note that I converted `freq_dict[word]` to a string in the `print` statement in order to concatenate a word with its frequency.

To summarize, the following code should do the trick:

```python
import sys
lines=sys.stdin.readlines()
freq_dict={}
for line in lines:
    words=line.strip().split()
```
```python
def test_zipf(words, freq_dict):
    for word in words:
        word = word.lower()
        if freq_dict.has_key(word):
            freq_dict[word] = freq_dict[word] + 1
        else:
            freq_dict[word] = 1
    print(' '.join(sorted(freq_dict.items(), key=lambda x: x[1], reverse=True))
```
We can use a dictionary to implement (5). That is,

- Initialize a dictionary of bins called `bin_dict`.
- Look up the word-frequency dictionary named `freq_dict` which we created for (1).
- For each word in `freq_dict`, do the following:
  - Retrieve its frequency from `freq_dict`.
  - If the retrieved frequency is already a key of `bin_dict`, append the word to its value.
  - Else, add the frequency as a key and list the word as its value.

More specifically:

```python
bin_dict = {}
for word in freq_dict.keys():
    frequency = freq_dict[word]
    if bin_dict.has_key(frequency):
        bin_dict[frequency].append(word)
    else:
        bin_dict[frequency] = [word]
```

We can implement step (6) as a function. We want the function to return the rank of a given word according to its frequency. So let’s name the function `freq_rank`. If you think about how we approached the problem, the function needs to know three things: what the word is, what the frequency of the word is, and how words are grouped into bins. That is, it takes three arguments: `word`, `frequency`, `bin_dict`. Here’s what I mean in the Python language:

```python
def freq_rank(word, frequency, bin_dict):
```

We want the function to implement steps (6-a) ~ (6-d). Here’s how:

```python
sum = 0
for bin_freq in bin_dict.keys():
    if bin_freq > frequency:
        sum = sum + len(bin_dict[bin_freq])
return sum + 1
```

You should realize that step (6-a), identifying which bin the given word belongs to is already captured by the word frequency. That’s because we labeled each bin according to the frequency of the words that belongs to the bin. The for-loop above is used to identify the bins that store words with higher frequencies and add the number of words that belong to those bins (6-b, 6-c). Recall that `len(x)`, where `x` is a list, returns the number of elements in `x`. The return statement at the end implements step (6-d).

OK. We took care of step (2) via (5) and (6). Compared to that, implementing (3) is almost trivial. We plug in a given word to the `freq_rank` function we just defined and multiply the function output by the word frequency. For example, if the word we’re interested in is `the`, we do the following:
freq_rank('the', freq_dict['the'], bin_dict) * freq_dict['the']

Step (4) is about comparing the product of word frequency and its rank across different words in a given corpus. I guess there are many ways to do this. But to simplify things, let’s just make our program print out the following four things for each word:

- rank
- word
- frequency
- frequency * rank

This can be implemented using a print statement. You should know how.

So here is how all the pieces are put together (also see /home/ling115/zipf.py):

import sys

# Read lines from the standard input.
lines = sys.stdin.readlines()

# 1. Get word frequencies.

freq_dict = {}  # For nickname
for line in lines:
    words = line.strip().split()
    for word in words:
        word = word.lower()
        if freq_dict.has_key(word):
            freq_dict[word] = freq_dict[word] + 1
        else:
            freq_dict[word] = 1

# 2. Group the words according to their frequency

bin_dict = {}  # For nickname
for word in freq_dict.keys():
    frequency = freq_dict[word]
    if bin_dict.has_key(frequency):
        bin_dict[frequency].append(word)
    else:
        bin_dict[frequency] = [word]

# 3. Define how to rank a word by its frequency.

def freq_rank(word, frequency, bin_dict):
    sum = 0
    for bin_freq in bin_dict.keys():
if bin_freq>frequency:
    sum = sum + len(bin_dict[bin_freq])
return sum+1

# 4. Test Zipf’s law.
for word in freq_dict.keys():
    word_freq = freq_dict[word]
    word_rank = freq_rank(word,word_freq,bin_dict)
    k = word_freq * word_rank
    print str(word_rank)+'	'+word+'	'+str(word_freq)+'	'+str(k)

Now try and test Zipf’s law with the Wall Street Journal corpus in /data/TREEBANK/RAW/WSJ/:

$ cat /data/TREEBANK/RAW/WSJ/*/* | python /home/ling115/zipf.py | sort -n | less