Mappings

Introduction

In this lab, we are going to write Python code to read the complete works of William Shakespeare and then report the word frequencies of the words used in his works.

Objectives

The purpose of this lab is to help you:

1. gain familiarity with class inheritance
2. learn the concept of doubling array size
3. implement and use the basic operations of a mapping (dictionary)
4. learn how to handle a file and practice some string operations
5. learn what to do when we need to sort a dictionary according to the values
6. learn how to implement hash functions and in particular good ones

The Complete Works of William Shakespeare

The text file that has the complete works of William Shakespeare is provided with the skeleton code. This text file has a special format. That is, every word or symbol is followed by a space. This makes the job of splitting words easier. This text file has all the works of Shakespeare, and it is very long. The size of the text file is about 4.5 Mbytes. The total number of lines is 129,107, the total number of words and symbols is 980,637, and the total number of characters is 4,538,523.

Here is the beginning of the text file:

A MIDSUMMER-NIGHT'S DREAM

Now , fair Hippolyta , our nuptial hour
Draws on apace : four happy days bring in
Another moon ; but O ! methinks how slow
This old moon wanes ; she lingers my desires ,
Like to a step dame , or a dowager
Long withering out a young man's revenue .

Four days will quickly steep themselves in night ;
Four nights will quickly dream away the time ;
And then the moon , like to a silver bow
New-bent in heaven , shall behold the night
Of our solemnities .

Go , Philostrate ,
Stir up the Athenian youth to merriments ;
Awake the pert and nimble spirit of mirth ;
Turn melancholy forth to funerals ;
The pale companion is not for our pomp .

Hippolyta , I woo'd thee with my sword ,
And won thy love doing thee injuries ;
But I will wed thee in another key ,
With pomp , with triumph , and with revelling .

We will read this text file and count the number of times each of the words and symbols appear in the text file.

Part 1: Solving the problem using Python dictionary (in-lab)

We first want to solve the problem using Python dictionary. To this end you are required to write a function called `getTokensFreq` that receives as a parameter a file name and returns a dictionary that includes all words as keys and their frequencies as the values.

Further, you are required to write a function called `getMostFrequent` that receives as parameters a dictionary `d` and the number of required frequent tokens `k`. The function returns a list of tuples of the `k` frequent key-value pairs in the dictionary.

Initially, read the text file into a string and split the string into symbols and words. And then loop through the list of symbols and words and use a dictionary to count the number of times these symbols and words appeared in the list. After that, use Python's in-built `sorted` function (https://www.programiz.com/python-programming/methods/built-in/sorted) to sort the dictionary according to the values and save the result into a list. Lastly, return the top `k` frequently used tokens and their corresponding frequencies.

Example:
Assume that the following text is stored in `f.txt`:
"I felt happy because I saw the others were happy and because I knew I should feel

happy, but I wasn’t really happy."

```
d = getTokensFreq('f.txt')
print(d)
# prints {'i': 5, 'felt': 1, 'happy': 4, 'because': 2, 'saw': 1,
#'the': 1, 'others': 1, 'were': 1, 'and': 1, 'knew': 1, 'should': 1,
#'feel': 1, ',': 1, 'but': 1, 'was': 1, 'not': 1, 'really': 1, '.': 1}

freq = getMostFrequent(d, 5)
print(freq)
# prints [('i', 5), ('happy', 4), ('because', 2), ('felt', 1), ('saw', 1)]
```

Part 2: Solving the problem using `HashMapping`

Your job now is to implement a class `HashMapping` and use this class to replace the dictionary above and repeat the above operations and obtain the same results. Note that you are provided with classes `Entry`, and `ListMapping`. You will need these two to implement the class `HashMapping`

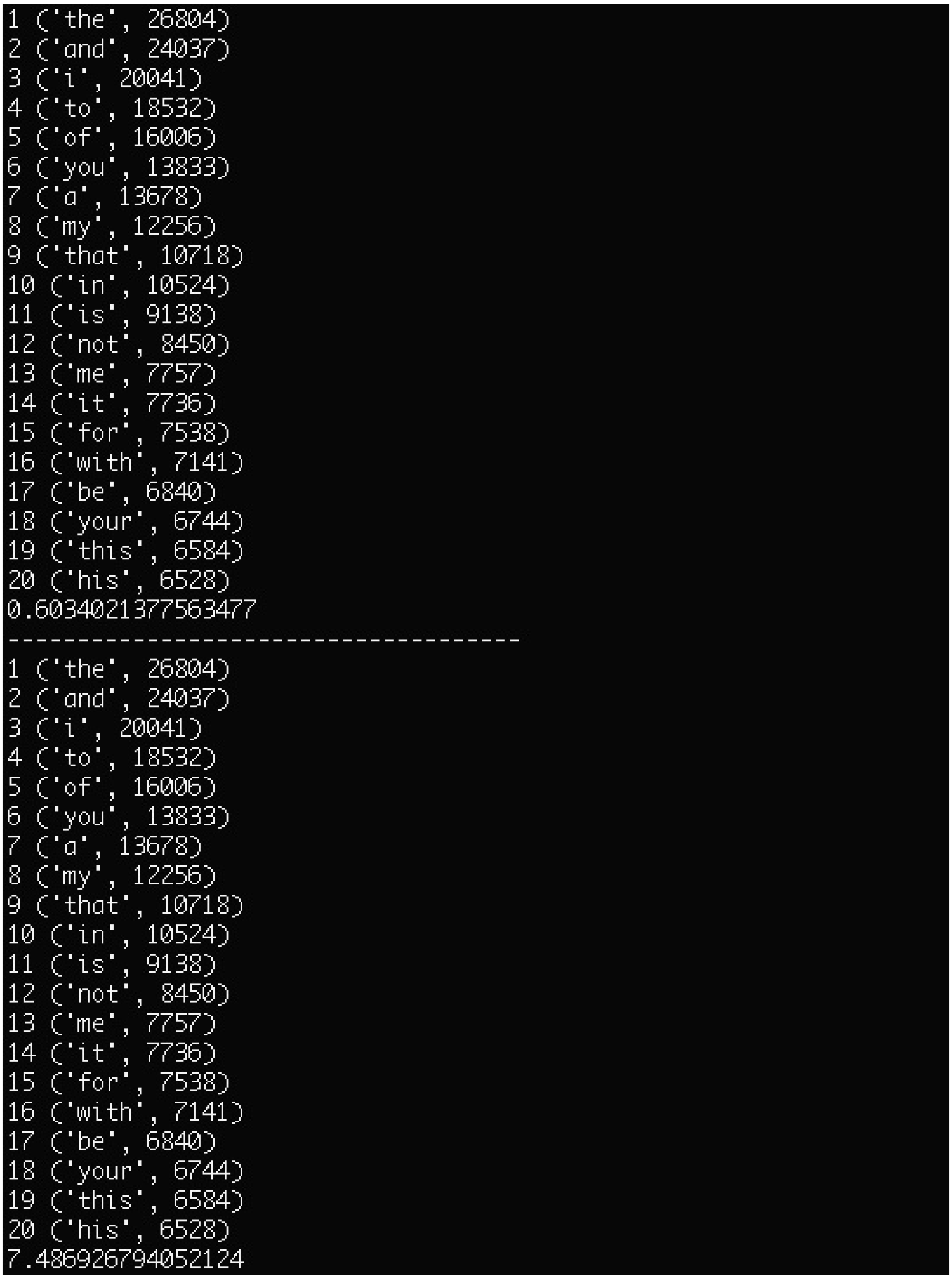
The `Entry` class is to store an entry of key-value pair. It has two attributes., `key` and `value`. `ListMapping` stores all the entries in a list. Its important methods are to get a value associated to a given key and to set a new value to a given key.

The `ListMapping` takes linear time because it has to iterate through the list. We could make this faster if we had many short lists instead of one large list. Then, we just need to have a quick way of knowing which short list to search or update.

We're going to store a list of `ListMappings`. The size of this list will be `1000`. For any key `k`, we want to compute the index of the right `ListMapping` for `k`. We often call these `ListMapping`s buckets. This term goes back to the
idea that you can quickly group items into buckets. Then, when looking for something in a bucket, you can check all the items in there assuming there aren’t too many.

This means, we want an integer, i.e. the index into our list of buckets. A hash function takes a key and returns an integer. Most classes in python implement a method called `__hash__` that does just this. We can use it to implement a mapping scheme that improves on the `ListMapping`. This is called `HashMapping` which you will be implementing.

Below is a screenshot of results for the top 20 words. Using dictionary, we get the job done in 0.6 seconds. Using `HashMapping`, it takes around 7.5 seconds.



```
1 ('the', 26804)
2 ('and', 24037)
3 ('i', 20041)
4 ('to', 18532)
5 ('of', 16006)
6 ('you', 13833)
7 ('a', 13678)
8 ('my', 12256)
9 ('that', 10718)
10 ('in', 10524)
11 ('is', 9138)
12 ('not', 8450)
13 ('me', 7757)
14 ('it', 7736)
15 ('for', 7538)
16 ('with', 7141)
17 ('be', 6840)
18 ('your', 6744)
19 ('this', 6584)
20 ('his', 6528)
0.6034021377563477
--------------------------------------
1 ('the', 26804)
2 ('and', 24037)
3 ('i', 20041)
4 ('to', 18532)
5 ('of', 16006)
6 ('you', 13833)
7 ('a', 13678)
8 ('my', 12256)
9 ('that', 10718)
10 ('in', 10524)
11 ('is', 9138)
12 ('not', 8450)
13 ('me', 7757)
14 ('it', 7736)
15 ('for', 7538)
16 ('with', 7141)
17 ('be', 6840)
18 ('your', 6744)
19 ('this', 6584)
20 ('his', 6528)
7.486926794052124
```

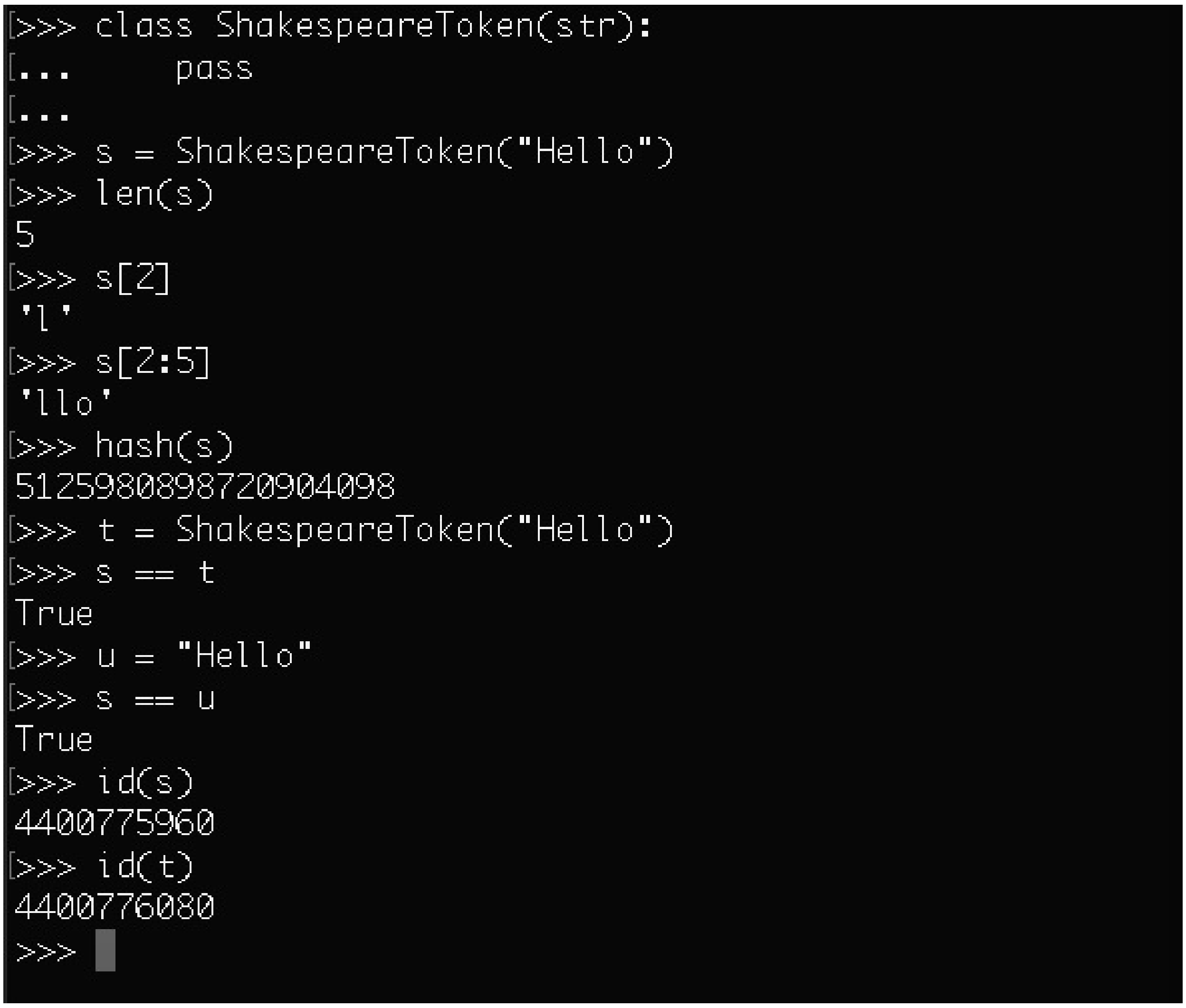
Let's see if you can do a better job implementing `HashMapping` than this.

Part 3: Shakespeare Tokens

In this part you are required to develop a class, called `ShakespeareToken`, that represents a Shakespeare token, which is really just a word from his text. Every token is a string so we can define `ShakespeareToken` class to inherit from Python string class which is called `str`. So the the definition of your class will look like:

```
class  ShakespeareToken(str):
    pass
```

This minimal definition of `ShakespeareToken` can be used in the same way as a string. For example, the following code works:



```
>>> class ShakespeareToken(str):
...     pass
...
>>> s = ShakespeareToken("Hello")
>>> len(s)
5
>>> s[2]
'l'
>>> s[2:5]
'llo'
>>> hash(s)
5125980898720904098
>>> t = ShakespeareToken("Hello")
>>> s == t
True
>>> u = "Hello"
>>> s == u
True
>>> id(s)
4400775960
>>> id(t)
4400776080
>>>
```

You can see that the hash method and the related comparison operator work as well.

Note: If you do not know why the above code works without having this functionality defined explicitly in `ShakespeareToken` class please learn more about inheritance before proceeding further.

Your job in this part is to develop your own hash method for the `ShakespeareToken` class. To do that you need to override the `__hash__` method. In this part the hash method that you are required to define is a “bad” one where it returns the length of the underlying token.

```
s = ShakespeareToken("Hello")
t = ShakespeareToken("Computer")

hash(s)         ## returns 5
hash(t)         ## returns 8
```

Part 4:

Write a second implementation of `ShakespeareToken` called `ShakespeareToken2` that improves the `__hash__` method. This method will just add up the ASCII values of the characters in the given string and return this sum. This method is case sensitive.

```
s = ShakespeareToken2("Hello")
print(hash(s))    ## prints 500

t = ShakespeareToken2("hello")
print(hash(t))      ## prints 532
```

Next, write a third implementation of `ShakespeareToken` called `ShakespeareToken3` that further improves the `__hash__` method. This method will use the following formula to calculate and return the hash of a string, `s` .

$$hashVal = s[0]_{ascii} * p^0 + s[1]_{ascii} * p^1 + s[2]_{ascii} * p^2 + \cdots + s[i]_{ascii} * p^i$$

where `p` is a prime number. `p` is usually the prime number closest to the length of the input alphabet. In our case, the number of upper case and lower case characters add up to 52. So the value of `p` will be 53. This method is also case sensitive.

```
s = SkakespeareToken3("Hello")
print(hash(s))          ## prints 892230904

t = SkakespeareToken3("hello")
print(hash(t))         ## prints 892230936
```

Part 5

The number 1000 that you use for size in part 2 is pretty arbitrary. If there are many many entries, then one might get 1000-fold speedup over ListMap, but not more. It makes sense to use more buckets as the size increases. Keeping track of the number of entries help in increasing the number of buckets. As the number of entries grows, we can periodically increase the number of buckets. Here is a method called `_double` that increases the number of buckets.

```python
def _double(self):
    # Save a reference to the old buckets.
    oldbuckets = self._buckets
    # Double the size.
    self._size *= 2
    # Create new buckets
    self._buckets = [ListMapping() for i in range(self._size)]
    # Add in all the old entries.
    for bucket in oldbuckets:
        for key, value in bucket.items():
            # Identify the new bucket.
            m = self._bucket(key)
            m[key] = value
```

It's not enough to just append more buckets to the list, because the `_bucket` method that we use to find the right bucket depends on the number of buckets. When that number changes, we have to reinsert all the items in the mapping so that they can be found when we next get them.

Write a class called `ExtendableHashMapping` that inherits `HashMapping` and it runs the `_double` method whenever the number of entries is more than the number of buckets. You will have to perform this check in `__setitem__` after setting the value to the given key.

Next, add a method called `statistics` to your `HashMapping` class (**NOT** `ExtendableHashMapping`) that prints out statistics about your `HashMapping` such as:

1. total number of buckets
2. how many buckets are empty
3. what is the size (list length) for the largest bucket
4. the average size
5. the standard deviation for bucket size

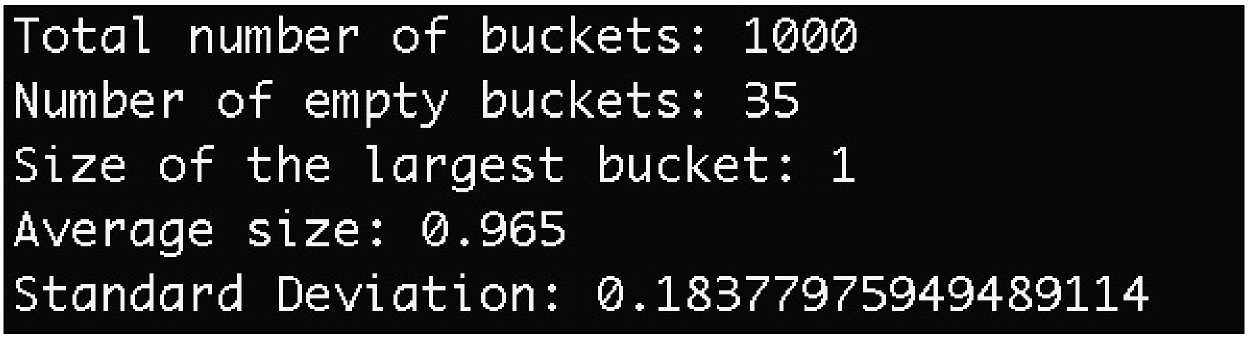
Note: For testing purposes, add a return statement that returns these 5 values in a tuple. Make sure to follow the order shown above. So your tuple should be of the form `(total_buckets,` `empty_buckets, largest_bucket, average_size, std_dev)`

Example:

```
map1 = HashMapping()
for i in range(965):        ## size is 1000
    map1[i] = 'i'

map1.statistics()
```

The output is as follows:



```
Total number of buckets: 1000
Number of empty buckets: 35
Size of the largest bucket: 1
Average size: 0.965
Standard Deviation: 0.18377975949489114
```

Now that you have the `statistics` method, I encourage you to try the following to see what the statistics look like.

```
import time

f = open('shakespeare.txt', 'r')
data = f.read()
f.close()

data = data.split()

start = time.time()
map1 = HashMapping()
for i in range(len(data)):
   key = data[i]
   s = SkakespeareToken(key)     ## try the other two ShakespeareToken classes
   if key not in map1:
       map1[key] = hash(s)%map1._size
end = time.time()

print("Statistics for HashMapping:")
print("---------------------------")
map1.statistics()
print("Time:", end - start)

start = time.time()
```

```
map1 = ExtendableHashMapping()
for i in range(len(data)):
   key = data[i]
   s = SkakespeareToken(key)    ## try the other two ShakespeareToken classes
   if key not in map1:
       map1[key] = hash(s)%map1._size
end = time.time()

print("\nStatistics for ExtendableHashMapping:")
print("-------------------------------------")
map1.statistics()
print("Time:", end - start)
```

You can try this out for the other two classes that you have written in part 4 as well.