Hash Map Word Count Project

Purpose: You may in the past have done word count exercises in Python using the builtin dict type. For this project, you will implement and use use a simplified key-value store like a dict called **HashMap** to do the same thing. The application will open the AliceInWonderland.txt file and parse the individual words from that file. Each word will be added to the Hash-Map with the get() and set() methods. The word will be the "key". Initially, that key will not be in the map, but get() must be defined to take a default value that is returned if a key is NOT in the map. Subsequently, when the same key is processed, get() will tell what the current count is and then your driver can use set() to increment the count by one.

Once the entire text of Alice In Wonderland has been added to the map, the 15 most frequently occurring words must be found and displayed. It should look like:

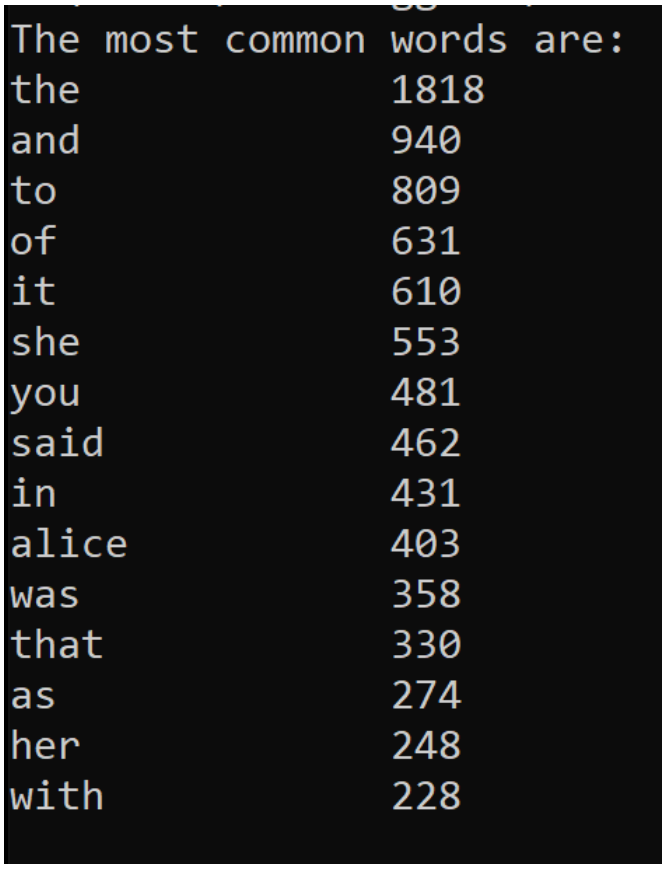

figure 1. Output of main() from main.py

Here is some code which will take a line of “raw” text and return a list of all words in the line. It converts the entire line of text to lower-case. It will discard all punctuation and only include a word into the output list if it has more than one letter. It is recommended that you use this code in main.py.

```
def clean_line(raw_line):
    '''removes all punctuation from input string and
    returns a list of all words which have a length greater than one '''
    if not isinstance(raw_line, str):
        raise ValueError("Input must be a string")
    line = raw_line.strip().lower()
    line = list(line)
    for index in range(len(line)):  # pylint: disable=C0200
        if line[index] < 'a' or line[index] > 'z':
            line[index] = ' '
    cleaned = "".join(line)
    words = [word for word in cleaned.split() if len(word) > 1]
    return words
```

HashMap ADT (hashmap.py)

A HashMap ADT supports the following operations:

- get(key [,default]): Return the value for *key* if *key* is in the dictionary, else *default*. If *default* is not given, it defaults to `None`
- set(key, value): add the (key,value) pair to the hashMap. After adding, if the load-factor >= 80%, rehash the map into a map double its current capacity.
- clear: empty the HashMap
- capacity: Return the current capacity--number of buckets--in the map.
- size: Return the number of key-value pairs in the map.
- keys: Return a list of keys.
- rehash: rebuild the table to reduce the load factor. The new table should be **twice** the capacity of the current table. Typically, this is used internally only.

Collision Resolution

The Hash-Map must use a "Linear Probe" approach with an **initial capacity of 8 buckets**. The number of buckets can dynamically grow, so the capacity is not fixed. See load factor.

Load Factor and Rehashing

A critical statistic for a hash table is the *load factor*, defined as $f = \frac{n}{k}$

where

- f is the load factor.

- *n* is the number of entries in the hash map.
- *k* is the number of buckets.

As the load factor grows larger, the hash table becomes slower, and it may even fail to work, depending on the method used, and the hashing function. The expected constant time property of a hash table assumes that the load factor be kept below some bound. Our solution here is to double the size of the number of buckets and rehash all the entries. **For this project, rehash when the load factor is >= 80%**

Test Cases

Following is the set of assertion-based test cases that your program must pass, and by which your code will be graded. *You will be given the pytest unit test code to run against your code as you develop it. This allows you to learn how test are written, and to know what your score is going to be when the code is graded before you submit it.*

Test If Key is Present

Create an empty HashMap. assert capacity() == 8, assert size() == 0

assert map get key "asdf" == None

assert map get key "asdf" with default 0 == 0

set("qwerty", 12345) assert get("12345") == 12345

assert size == 1

Test Nominal Map

keys = ["first", "second", "third", "fourth", "fifth", "sixth", "seventh", "eighth", "ninth", "tenth"] values = list(range(1,11))

add keys with corresponding values to HashMap

assert map.get("sixth") == 6 assert map.get("sixth") != 7) assert map.get("sixth") != 5

assert map.size() == 10

assert map.capacity() == 16

map.set("third") = 409 assert map.get("third") == 409

Test Rehash

keys = ["first", "second", "third", "fourth", "fifth", "sixth", "seventh", "eighth", "ninth", "tenth"] values = list(range(1,11))

create empty HashMap. assert hm.size() == 0 assert hm.capacity() == 8

add first 5 keys with corresponding values to HashMap. assert hm.size() == 5 assert hm.capacity() == 8

output = hm.keys() assert len(output) == 5

asset every element in output is in keys

add rest of keys with corresponding values to HashMap. assert hm.size() == 10

assert hm.capacity() == 16 (adding the second 5 keys should have caused a rehash, which would have doubled the capacity)

output = hm.keys()

assert every element in output is in keys

assert every element in keys is in output

Test Main Driver

assert then the output of main() looks similar to figure 1 above.

Test Coding Standards

Running PyLint on main.py and hashmap.py must rate your code at an 8.5 or higher

Grading (100 points)

Grades will come from the unit tests.

- Key Is Present 10
- Nominal Map 35
- Rehash 10
- Main Driver 35
- Coding Standards 10

Files to turn in through Canvas

- main.py
- hashmap.py