LEARNING AI-DS-ML SKILLS

# PYTHON PROGRAMMING

## CHAPTER 5 DICTIONARIES AND HASHING

BY FRED ANNEXSTEIN, PHD

# PYTHON PROGRAMMING

# 5. DICTIONARIES AND HASHING

BY FRED ANNEXSTEIN, PHD

# CHAPTER 5 OVERVIEW

In this module we will be covering the important concept of searchable hash tables used for efficient data lookups. We show how to use the built-in python datatype called a dictionary to construct them. Dictionaries are one of Python's most valued features, and they are the key datatype for many efficient and elegant algorithms. We show how to use hash tables for frequency tables and for natural language models and we construct an AI chatbot based on such tables.

# CHAPTER 5 OBJECTIVES

By the end of this module, you will be able to...

- Understand the syntax and methods of python dictionaries.
- Apply and implement tables such as frequency, histograms, and hash tables from python dictionaries.
- Construct a recursive function that uses memoization.
- Apply successor table dictionaries in the construction of a natural language model
- Implement a chatbot application using dictionary-based successor table.
- Introduce Markov chains - a table in which the set of values is assigned a likelihood or probability.

# DICTIONARY SYNTAX

Consider storing a collection of key/value pairs. Keys are used to locate table records which contain the values of interest. Python's efficient key/value table structure is called a dictionary or simply a ‘dict’. The contents of a dict can be written as a series of key:value pairs within braces { } surrounding, as follows:

```
dict1 = {key1:value1, key2:value2, ... }
```

We contract an "empty dict" with just an empty pair of curly braces.

```
dict1 = {}
```

Looking up or setting a value in a dict uses square brackets, for example, the following statement looks up the value associated with the key ‘foo'.

```
dict1[‘foo’]
```

Strings, numbers, and tuples all work as keys, and an object of any type can be a value. Other types may or may not work correctly as keys (strings and tuples work cleanly since they are immutable).

Looking up a value which is not in the dict throws a KeyError. You can use "in" operator to check if the key is in the dict, or use statement `dict.get(key)` which returns the value paired with key, or returns None if key is not present. Adding another argument will allow you to specify what value to return in the case of key not-found.

Let us look at a code block which we use a dictionary to build a hash table, starting with the the empty dict {}, and storing key/value pairs adding one at a time to the table:

```
dict1 = {}
dict['a'] = 'alpha'
dict['g'] = 'gamma'
dict['o'] = 'omega'

>>> dict1
{'a': 'alpha', 'g': 'gamma', 'o': 'omega'}

>>> dict1['a']
‘alpha’

>>> dict1['b']
Traceback (most recent call last):
KeyError: ‘b'

>>> dict1[‘b'] = 6

>>> dict1[‘b'])
6
>>> 'b' in dict1
True
```

# HISTOGRAMS

Dictionaries are useful for computing the frequency of values in a collection. Here is an example that will work with any sequence type including strings.

```
def frequencies(c):
   freq = {}
   for i in c:
       if i not in freq:
           freq[i] = 1
       else: freq[i] += 1
   return freq

>>> frequencies('hello')
{'h': 1, 'e': 1, 'l': 2, 'o': 1}
```

The first line of the function creates an empty dictionary. The for loop traverses the string. Each time through the loop, if the character is not in the dictionary, we create a new item with key and the initial value 1, since we have now seen this letter once. If i is already in the dictionary, then we increment the counter value.

# FOR LOOPS

You can use a for-loop with a dictionary, and we therefore say that a dictionary is iterable. This iteration traverses the keys of the dictionary. For example, print_dict(d) prints each

key and the corresponding value in the dictionary d.

```
def print_dict(d):
   for c in d:
       print(c,':' d[c])

>>> h = frequencies('rabbitt')
>>> print_dict(h)
r:1  a:1  b:2  i:1  t:2
```

## MEMOIZATION

Recall the recursive definition of the Fibonacci function from Chapter 3. We saw that the tree recursion meant that the function would branch so much that the function will not finish for even modest argument values.

One solution to this problem is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored in the dictionary for later use is called a memo. Here is a "memoized" version of Fibonacci function. We use name 'known' for a dictionary that keeps track of the Fibonacci numbers we already know. It starts with two items: 0 maps to 0 and 1 maps to 1.

```
known = {0:0, 1:1}

def fib(n):
    if n in known:
        return known[n]
    result = fib(n-1) + fib(n-2)
    known[n] = result
    return result

>>> for i in range(1,6):
        print(i,':',fib(i), end='  ')

1 : 1   2 : 1   3 : 2   4 : 3   5 : 5
```

Whenever the fib function is called, it checks the known dictionary. If the result is already there, it can return immediately. Otherwise it has to compute the new value, add it to the dictionary, and return it. If you run this version of fib and compare it with the original recursive version, you will find that it is much faster for large arguments.

## SUCCESSOR TABLES

We now consider the problem of building a hash table for a natural language model, and use that model to create a bot for Twitter. The natural language model we use is called the bigram word model. That is, for every word in a text file or corpus of files we will store all possible successor words. These pairs will be stored in a "successor table" implemented using Python dictionaries.

Once we are done building the successor table we can generate a sentence or 'tweet' by starting with a (random) word and then randomly choosing a word from the successor list. Then we repeat the processes to look up the next successor word, and so on. This process eventually will terminate in a period (".") or other terminal character.

The word successor table is just hash table and is implemented using an ordinary Python dictionary. The keys in this dictionary are (predecessor) words, and the values are lists of all successor words found in the corpus.

Here is the definition of a function to build_successors_table. The input argument tokens is a list of words corresponding to a given corpus text that is split into “tokens" or individual words. The output is a successors table. (By default, we assign the first word is a successor to "."). See the example below.

```
def build_successors_table(tokens):
    table = {}
    prev = '.'
    for word in tokens:
        if prev not in table:
            table[prev] = []
        table[prev] += word
        prev = word
    return table
```

# GENERATING RANDOM SENTENCES

Let us generate some sentences at random simply by using the successor table. Suppose we are given some starting word. We can look up this word in our table to find its list of successors, and then randomly select a word from this list to be the next word in the sentence. Then we just repeat the process until we reach some ending punctuation.

To randomly select from a list, first make sure you import the Python random library with `import random` and then use the expression `random.choice(my_list))`

```
def random_sentence(word, table):
    """Returns a string that is a random
sentence starting with word, and choosing
successors from table.
    """
    import random
    result = ' '
    while word not in ['.', '!', '?']:
        result += word + ' '
        word = random.choice(table[word])
    return result + word
```

# MARKOV CHAINS

In our successor table example above, we used a list of words as our values. In our random sentence application we treated each word equally by making a random choice when we generated new sentences. We can get more powerful language and data models by allowing bias which associates a belief likelihood or probability weight to each element of the list of values. These weights bias the possible outcomes to be more probable choices. This is the idea of a Markov chain - a table in which the set of values is assigned a likelihood or probability.

Here is a simple example script for a Markov Chain modeling three states of the daily weather. We construct a Markov chain by associating a successor probability with each possible day state. We represent the weights by enhancing our dictionary where we let table values be lists of pairs stored as tuples. Each tuple represents a daily weather state along with its probability as successor state. Note that in our example below when it is ‘Sunny’ there is equal 1/3 chance that the next day will be any weather state. When it is ‘Cloudy’, there is 1/2 chance it will rain tomorrow and equal chance 1/4 it is sunny or cloudy. But when it is ‘Raining’ the chance of rain the next day is 3/5, and equal chance 1/5 is sunny or cloudy.

```
import random
s,c,r = 'Sun', 'Clouds', 'Rain'
weather = [s,c,r]
weather_table = { s : [(s,1/3),(c,1/3),(r,1/3)],
                  c : [(s,1/4), (c,1/4), (r,2/4)],
                  r : [(s, 1/5), (c,1/5), (r, 3/5)]}
```

Here weather only depends on the previous day and the probabilities associated with that weather state is stored as a list of tuples in the table. Here is a randomly generated 10-day forecast based on this Markov Chain model.

```
nextday = random.choice(weather)
for day in range(10):
   weatherlst = weather_table[nextday]
   print(s,c,r)
   probs = [w[1] for w in weatherlst]
   print(probs)
   nextday = random.choices(weather, probs, k=1)[0]
   print("Day", day, ":  ", nextday)

Sun Clouds Rain
[0.25, 0.5, 0.25]
Day 0 :   Clouds
Sun Clouds Rain
[0.25, 0.5, 0.25]
Day 1 :   Clouds
Sun Clouds Rain
[0.25, 0.5, 0.25]
Day 2 :   Rain
Sun Clouds Rain
[0.2, 0.2, 0.6]
Day 3 :   Sun
Sun Clouds Rain
[0.333, 0.333, 0.333]
Day 4 :   Sun
```

Note that we are passing to the **`random.choice`** function an optional parameter representing Markov Chain weights which we name **`probs`**. These probabilities were extracted or projected from the weather_table using a list

comprehension to extract just a list of numbers to be used as an argument.

## CHAPTER EXERCISES

**Exercise 1.** Write a function that reads the words in the file words.txt and stores them as keys in a dictionary. It doesn't matter what the values are. Then you can use the in operator as a fast way to check whether a string is in the dictionary.

**Exercise 2.** Memoize the Ackermann function, see the definition at https://en.wikipedia.org/wiki/Ackermann_function

**Exercise 3.** Use a dictionary to write a fast predicate function called **`has_duplicates(seq)`** which takes an argument that is a sequence, and determines whether or not there are any duplicate values in the sequence.

**Exercise 4.** The "rank" of a word is its position in a list of words sorted by frequency: the most common word has rank 1, the second most common has rank 2, etc.

Zipf's law describes a relationship between the ranks and frequencies of words in natural languages (http://en.wikipedia.org/wiki/Zipf's_law). Specifically, it predicts that the frequency, f, of the word with rank r is: $f = cr^{-s}$, where s and c are parameters that depend on the language and the

text. If you take the logarithm of both sides of this equation, you get the equation:

$$\log f = \log c - s \log r$$

Thus if you plot log f versus log r, you should get a straight line with slope –s and intercept log c.

Write a program that reads a text from a file, counts word frequencies, and prints one line for each word, in descending order of frequency, with log f and log r. Use the graphing program of your choice to plot the results and check whether they form a straight line. From the plot can you estimate the value of s?