Part 2: Tokenisation

Topics:

- Lists
- Dictionaries
- Iteration
- Reading a file
- Split function
- Tokenisation
- Frequency counts

Research projects that make use of *Text and Data Mining* often base many of their analyses on counts of the words that appear within a text. Such frequency counts begin with a process in which the text is divided into its individual words. This preparatory process is generally referred to as "tokenisation". The individual words that are found through this process are also named the "tokens". This section of the tutorial explains how texts may be tokenised using Python, and how the tokens that are produced in this way can be counted. Before this can be done, however, a number of other techniques need to be explained, including the principle of working with lists and with dictionaries.

Lists

The variables that were discussed in the first part of the Python tutorial were all assigned single values, such as strings or integers. Alternatively, it is also possible to work with variables that contain collections of values. One example of such a variable that can hold multiple values is the list. Lists can be created by surrounding all the values that you want to gather by square brackets. The individual values need to be separated by commas. The individual values in such lists are referred to as elements. The statement below creates a list containing the five days of the working week:

```
week = [ "Monday" , "Tuesday" , "Wednesday" , "Thursday" ,
"Friday" ]
```

The elements in the list above are all strings. The elements do not all have to be of the same type, however. It is also allowed to mix strings with integers, for instance.

When lists are created in this manner, Python numbers the elements in the array automatically, following the order in which these values given. These numbers will function as indices that can be used to access the individual items in the array. Bear in mind that Python starts counting at 0. In the example above, the element "Monday" will have key "0", and the element "Thursday" will have the value "3". As is the case for strings and their individual characters, the separate elements in the list can be accessed using the bracket operator.

Another similarity between lists and strings is that the lengths of both types of both types of variables can be found using the len() function. In the case of lists, len() returns the number of elements in the list.

```
week = [ "Monday" , "Tuesday" , "Wednesday" , "Thursday" ,
"Friday" ]

print( week[3] )
# This prints "Thursday"

print( week[-1] )
# This prints "Friday"

print( len(week) )
# This prints 5
```

Listing 2.1.

The append() method can be used to add items to an existing list. Note that methods, in contrast to functions, need to be pasted onto variables using the period. The following code can be used to add the weekend to the week variable that was discussed above.

```
week.append("Saturday")
week.append("Sunday")
```

The remove() method may be used to delete a specific item from the list. In the parentheses, you need to mention the item that needs to be removed.

```
week.remove("Monday")
```

Finally, the items in a list can be arranged alphabetically or numerically by making use of the sort() method. The code below offers an illustration.

```
unsorted = [ 5 , 8 , 2 , 9 , 1 , 8 ]
unsorted.sort()
print( unsorted )

## The output is as follows:
## [1, 2, 5, 8, 8, 9]
```

Listing 2.2.

Dictionaries

A second data structure which can be used to collect multiple values is the dictionary. While the indices of lists consist of numbers which are assigned automatically, the indices of dictionaries can be any type of value, and programmers also need to define these indices

themselves. Dictionaries can be used to unite two values which have a close connection, such as a numerical value and a textual label for this value. Unlike the elements in a list, the elements in the dictionary are not ordered.

A dictionary can be created using the dict() function. This function results in an empty dictionary. Elements can be added to the dictionary by using the bracket operator containing the index. Importantly, each of these indices must be unique.

```
1   data = dict()
2
3   data['firstName'] = 'William'
4   data['lastName'] = 'Shakespeare'
5   data['yearOfBirth'] = 1564
```

Listing 2.3.

Dictionaries can be applied usefully in word frequency lists. The index, in this case, can be the word, and the value can be the number of times this particular word occurs in a text. Lines 3 and 4 in listing 2.4. below, for instance, state the number of times the words 'the' and 'of' occur in a certain text. In realistic text mining programmes, frequency values are not typed in manually, but calculated, as will be illustrated later.

Line 6 of this program prints the frequency of the word 'the'. As is the case for the elements in lists, individual dictionary items can be accessed using their indices, mentioned in the bracket operator.

```
1   frequency = dict()
2
3   frequency['the'] = 5790
4   frequency['of'] = 5634
5
6   print( frequency['the'] ) ;
7   ## This will print 5790
8
9   print( frequency.get( 'new' , 0 ) )
10  ## This will print 0
```

Listing 2.4.

As was mentioned, the individual values in dictionaries can be accessed by using their indices. When you mention an index that does not actually exist, however, the code produces an error. To avoid such error messages, you can make use of the get() method, as exemplified on line 8 of the code above. The get() method requires two values. Firstly, you need to mention the index of the element that you want to retrieve. Secondly, you have to provide a default value which should be returned when the index that was supplied does not exist. The statement on line 9 attempts to print the frequency of the word 'new', but as this index has not been added to the dictionary yet, the default value of zero is shown.

Iteration: while and for

As explained in Part 1 of the Python tutorial, flow control structures basically determine if and how many times certain statements needs to be executed. Repetitions of statements are generally referred to as iterations. In Python you can determine the number of times a set of statements are repeated by making use of 'while' or 'for'. As is the case for 'if', the 'while' keyword should be followed by a test, or, more precisely, by a Boolean expression. The 'while' keyword initiates a loop. The statements that follow 'while' will be repeated as long as the expression is true. Clearly, it is necessary to ensure that the expression can also be evaluated to 'false' at some point, since the repetition will continue endlessly otherwise. The variable that can end the loop is called the iteration value.

The code in listing 2.5. prints the multiplication table for the number 7. The number of iterations is determined by the variable named i. The value of this variable is incremented in each repetition. The loop will terminate after the variable i has reached the value 11.

```
1   number = 7
2   i = 1
3
4   while i <= 10:
5       print( str(i) + " times " + str(number) + " is "
         + str( i * number ) )
6       i += 1
```

Listing 2.5.

In line 6 of this code, the variable named i is incremented with value 1. The statement that is used is a shorthand notation for the following statement:

```
    i = i + 1
```

The keyword 'for' can be used effectively to navigate across all the elements in a list or in a dictionary. As can be seen in listing 2.6., 'for' needs to be used in combination with 'in' for this purpose. Next to the name of the list or the dictionary whose elements you want to retrieve, you also need to supply the name of a new variable. In the example below, this new variable is named 'day'. During each cycle, the value of the variable 'day' will be different; it will consecutively be assigned the various elements in the list named 'week'. The loop will continue as long as there are items in the list.

```
1   week = [ "Monday" , "Tuesday" , "Wednesday" , "Thursday" ,
    "Friday" ]
2
3   for day in week:
3       print( day )
```

Listing 2.6.

Reading a file

Research project that make use of *Text and Data Mining* often produce quantitative data about text corpora. The texts in such collections are generally stored as separate files on a disk. A basic initial operation, therefore, is to open a file and to read the contents of this file. In Python, files can be 'read' using the open() function. The result of this function is a new object which is called a file handler (or, more specifically, a TextIOWrapper object). Simply put, a file handler is an object which establishes a connection to the text file on your disk. You are free to name the file handle as you like. This file handler can access the text line by line. Text files typically use the hard return or the newline character to delineate lines or paragraphs. In practice, the open() function splits the text file into smaller units using the hard returns that occur in the file. The 'for' keyword can be used to iterate across the various units represented by this file handler. Listing 2.7. demonstrates how you can read and display the full contents of a text file, line by line.

When you use the open() function, you are also recommended to specify the character encoding scheme that has been used in the text file, using the 'encoding' parameter. This will help Python to process all the characters correctly.

```
1   text = open( "Prufrock.txt" , encoding = 'utf-8' )
2
3   for line in text:
4           print(line)
```

Listing 2.6.

In the case of short texts, you can also make use of the read() method of the file handler. When you do this, the entire text will not be divided into smaller units. The whole text will become available as one long string.

Tokenisation

The final thing that you need to know, before you can write a basic program that can tokenise a text, is that you can use the split() method to break a given string down into smaller units. Within the parentheses of the split() method, you need to provide a short string which can act as a separator. Since the words of a text are typically delineated by spaces, you can often use the space character (" ") as a delineator for the split() method. The result of this method is an ordered list which contains the words that can be distinguished in this way.

```
1   line = "If music be the food of love, play on"
2
3   words = line.split()
4
5   print( words[0] )
6   ## "If"
```

```
7
8  print( words[7] )
9  ## "play"
```

Listing 2.7.

Listing 2.8. combines the various techniques that have been discussed in this tutorial to create a simple tokeniser.

```
1   freq = dict()
2
3   text = open("Prufrock.txt" , encoding = 'utf-8' )
4
5
6   for line in text:
7
8       line = line.lower()
9       words = line.split(' ')
10      for w in words:
11          freq[w] = freq.get( w ,0 ) + 1
12
13  for word in freq:
14      print( word + " => " + str( freq[word] ) )
```

Listing 2.8.

Line 1 firstly creates an empty dictionary named 'freq'. This dictionary will be used to store the frequencies of the word in the text. Line 3 reads the contents of the file named 'Prufrock.txt' using open(). It assigns the contents of this text to a file handler named text. Next, the file is read line by line in the loop which is initiated using the 'for' keyword on line 6. During each cycle of the 'for' loop, the variable named 'line' represents a different line from the text file.

Line 8 converts all the characters in the various lines to lower case, to make sure that 'The' and 'the', for instance, are treated as the same word, despite that the fact that they contain characters in a different case. On line 9, the split() method is applied to this line. In this method, the space character is used as a delimiter. This has the effect that all the chunks that are separated by spaces will be recognized as separate units> These units (i.e. the words) will be placed in a list named 'words'. Lines 10 and 11 navigate across all of these words. The words that are found in this way are used as indices in the 'freq' dictionary.

The value associated with this index will be incremented using the get() method of the dictionary. The use of get() is convenient in this situation. The first time a word is found, it has not been defined any index yet in the 'freq' dictionary. As the index is not available at that point, the method returns the default value that is provided, namely zero. While Python reads the text, and as more words are found, the frequency values associated with the various words will be updated.

At the end of this program, the indices in the dictionary represent all the unique words that occur in the text. The values associated with these keys are the number of times these words occur in the text.

The final lines of the program prints a list of the words that were identified, together with their frequencies.

Writing to a file

When you run the code listing 2.8, the full output will be printed on the Command Prompt. When there are many lines to print, as in this particular example, it can be very difficult to read the output. In such cases, it can useful to create a new text file which will receive all the output. The results of the program can then be inspected by opening this new file in a text editor.

The function open(), which you have used to read files, can also be used to create a new file. Instead of referencing a file which already exists on your disk, you need to enter a new file name. Secondly, you also need to supply a second value, the character "w", which stands for "write". This "w" character makes it clear to Python that you want to write to a file. The open() function used with the "w" parameter similarly creates a file handler. This handler has a write() method, which functions very similarly to the print function. The crucial difference, however, is that the output is not sent to the Command Prompt, but to the file associated with this file handler.

It is also good practice to close the file when there is nothing left to print, using the close() method.

```
1   out = open('data.txt' , 'w')
2
3   out.write( "This text is in a file named 'data.txt' " )
4
5   out.close()
```

Listing 2.9.

Sorting a dictionary

Listing 2.8 does not sort the output in any way. The code simply prints the words and their frequencies in the order in which they were found in the file. To be able to analyse the most frequent words, it can be useful to sort the dictionary, however either by index, or by the value associated with this value.

When you have created a dictionary named freq, using the code in listing 2.8, it can be sorted by index using the following code. This code makes use of the in-built sorted function.

```
for word in sorted(freq):
    print( word + " => " + str( freq[word] ) )
```

Sorting by value is slightly more complicated. One of the most convenient ways is by making use of a so-called lambda function. The 'freq' dictionary that was used in examples above can be sorted by value, in a descending order, by making use of the code below. The details of this code shall not be explained in this tutorial. To sort dictionaries by value, you can simply

copy the code that is given here. Remember, however, that you need to provide the name of the dictionary to be sorted twice in the statement on line 1, and once in the statement on line 4. Using these statements, the most frequent words will be shown first. If the dictionary needs to be sorted in an ascending order, the reversed() function needs to be removed.

```
1   sorted = reversed( sorted( freq , key=lambda x: freq[x]) )
2
3   for w in sorted:
4       print ( w + ' => ' + str( freq[w] ) + "\n")
```