Part 3: Regular Expressions

Topics:

- Functions
- Classes
- Modules
- Regular Expressions
- Reading the contents of a folder

In Part 2 of this tutorial, you have learnt how you can create an application which can divide a text in its individual words, and which can calculate the frequencies of these words. Using the relatively simple method that was discussed, however, the data that was produced still contained quite a large degree of noise. If a word is followed by punctuation, for instance, this word was treated differently from a word that is surrounded only by spaces. Such challenges can be addressed effectively by making use of regular expressions. Python has an in-built module named 're', which enables programmers to manipulate strings in highly advanced ways. Before we can explain regular expressions, the concept of working with methods and modules needs to be clarified first.

Functions

Image that you need to write a program in Python which can calculate the final grades for a university course. Such grades are normally calculated using a fixed formula (e.g. a mid-term assignment which counts for 40% and a final exam which counts for 60%). To calculate the final grades, this formula needs to applied repeatedly with different values, as the students in the course evidently earn different marks. In such a situation, it would be quite inefficient if you simply repeated the code needed for making these calculations for every single student. In most programming languages, fortunately, it is also possible to reuse fragments of code by defining these as functions.

A function is essentially a set of statements which can be addressed collectively via a single name. Python has a number of in-built functions, such as print() or split(). Such functions represent certain actions. To have these actions executed, you need to refer to the names of these functions in your programs. Next to working with in-built functions, it is also possible to write your own functions. Working with functions can often be very effective. It enables you to decompose your larger problem into sub-problems and into units which can be reused as often as is needed. As illustrated in the code below, functions can be created using the 'def' keyword.

```
def calculateFinalMark( midTerm, exam ):
    finalMark = 0.4 * midTerm
    finalMark += 0.6 * exam
    return finalMark

```

```
 6   print( str( calculateFinalMark( 8 , 9 ) ))
 7   print( str( calculateFinalMark( 4 , 10 ) ))
 8   print( str( calculateFinalMark( 6 , 7 ) ))
```

Listing 3.1.

In listing 3.1., the code on lines 1-4 create the function calculateFinalMark(). The def keyword, on line 1, needs to be followed by the name of the function, which you can specify yourself. The name of the function also needs to end in a set of parentheses. Within these parentheses, you can optionally mention the values the function should operate on. These values mentioned within the parentheses are called the parameters of the function. If there are two or more parameters, these need to be separated by commas. In the example above, a certain calculation is carried out within the function, and the result of this calculation is mentioned after the keyword 'return'.

Once the function has been defined, it can be used, or invoked, on other locations in your program, as can be seen on lines 6-8 of listing 3.1. The print statements on these lines will show the result of the calculation that is returned by the function, using the values that are mentioned within the parentheses as a basis.

Classes

Python is based on a programming paradigm which is known as object-oriented programming. This paradigm, in short, involves an organisation in which variables and functions which are closely related can be brought together in structure known as a class. In the example below, the variables named 'title' and 'isbn', and the function named 'describe' are brought together in a class named 'Book'. A class can also be instantiated. An instance of a class is called an object, and this object can be given a new name. For this newly created object, all the variables and the all functions which are defined in the class can also be accessed, by appending the names of the variables or functions to the dot following the object name. This notation is called the 'period syntax'. Functions defined within classes are referred to as methods. A class thus functions as a blueprint or as a template for new types of objects.

```
 1   class Book:
 2       def __init__(self, title, isbn ):
 3           self.title = title
 4           self.isbn = isbn
 5
 6       def describe(self):
 7           print("Title: " + self.title)
 8           print("ISBN: " + str(self.isbn))
 9
10   title1 = Book("A Room with a View", "978-1420925432")
11   title1.describe()
```

Listing 3.2.

In the code above, line 10 creates a new object based on the class Book, named title1. Once the class Book has been instantiated in this way, the object based on the class specifications can make use of all the methods of the class. Line 11, for instance, uses the describe() function of the Book class.

Modules

A module, thirdly, is essentially a file with code that you can reuse across different programs. Modules usually contain functions or classes. While there are many in-built modules in Python, you can also create your own modules. A module must be saved using the .py extension. The name of the module is simply the name of the file without the extension.

The functions that are defined within modules can be imported into another program using the ‘import’ keyword. If you have created a module named ‘textMining.py’, for instance, all of its function can be imported into another program using the statement below:

```
import textMining
```

If this module contains a function named tokenise(), this function can be made available as follows.

```
tokens = textMining.tokenise()
```

Alternatively, it is also possible to import individual functions from a module.

```
from textMining import tokenise
```

This second way of importing code has the advantage that it is no longer necessary to use the period syntax. The function can then be used without referencing the name of the module:

```
tokens = tokenise()
```

Regular expressions

The standard installation of Python includes a useful module called ‘re’, which can be used to search texts on the basis of regular expressions. A regular expression is essentially a sequence of characters which define a pattern to search for in a text. The pattern can consist of actual characters or of generic representations of specific types of characters.

To work with the module, you firstly need to import it. The module ‘re’ contains a function called search(), which minimally requires two parameters. The first parameter is the pattern to search for, and the second parameter is the text string in which you want to search. The method returns the value ‘true’ if the pattern which is mentioned occurs in the string which is provided as the second parameter. The listing below offers an example.

```
import re

sentence = 'Mrs. Dalloway said she would buy the flowers herself.'
```

```
if re.search( 'flower' , sentence ):
    print('The pattern was found in the sentence!')

```

Listing 3.3.

In listing 3.2., the regular expression is simply a sequence of characters. It is also possible to work with more generic representations of characters, called character classes. The following character classes may be used:

.	Any character, except the newline.
\w	Any alphanumeric character: all 26 alphabetical characters, both in upper case and in lower case, all numbers and the underscore.
\d	Digits
\s	White space: the space, a tab or a newline character.
[...]	If classes such as \w or \d are too broad, and if only a limited number of characters are allowed on a specific position in a string, this set of possible characters may be supplied in square brackets.

You can also use quantifiers to specify the number of times a character or a pattern should occur.

{n,m}	Pattern must occur a least n times, at most m times
{n,}	At least n times
{n}	Exactly n times
?	Is the same as {0,1}
+	Is the same as {1,}
*	Is the same as {0,}

The code below offers a number of examples of regular expressions containing such character classes and quantifiers.

```
import re

sentence = "Keats's 'Ode on a Grecian Urn' was written in 1819."

if re.search( r'\d{4}' , sentence ):
    print('Found')
## Matches '1819'

```

```
 9   if re.search( r'U.n' , sentence ):
10       print('Found')
11   ## Matches 'Urn'
12
13   if re.search( r'K[aeuio]{2}t' , sentence ):
14       print('Found')
15   ## Matches 'Keat'
16
17   hits = re.findall( r'[aeuio]n' , sentence )
18   for h in hits:
19       print(h)
20   ## Four matches: 'on', 'an', 'en' and 'in'
```

Listing 3.4.

In the listing above, all the regular expressions are preceded by the character ‘r’, which, in this context, indicates that the strings defining the regular expressions make use of the ‘raw string’ notation. In short, it means that all characters need to be read literally, in their ‘raw’ form.

Line 17-20 above also illustrate the function of the findall() function from the ‘re’ module. This function creates a list containing all fragments from the string that is searched that match the regular expression.

Finally, you can also use anchors in regular expression. Anchors do not represent actual characters, but only locations within strings.

\b	A word boundary
^	The beginning of a string
$	The end of a string

A word boundary is a location in which an alphanumeric character is placed next to a character which is not an alphanumeric character, such as punctuation, a space or a new line character.

if you add the text “re.IGNORECASE” as the third parameter of the search() function, the search will take place in a case-insensitive manner. For examples of case-insensitive searches using word boundaries, see the listing below.

```
import re

line = "The sun shone, having no alternative, on the nothing new."

if re.search( r'\bSUN\b' , line , re.IGNORECASE ):
    print('Found')
# Matches 'sun', even though the pattern contains upper case

hits = re.findall( r'\bno\b' , line )
```

```
for h in hits:
    print(h)
## Matches 'no', but not 'nothing', as 'no' is followed by
## an alphanumeric character in the latter word
```

Listing 3.5.

In some cases, it may be necessary not only to test if a string contains a pattern, but also to retrieve the characters in the string that are matched by the regular expression. This can be accomplished by working with parentheses in the regular expression. These parentheses will have the effect that the characters in the string which match the regular expression are saved using the method group(). The parameter '1' of this method corresponds to the characters that have been matched. Listing 3.4. illustrates this principle. It is a program which can extract the direct speech from a longer sentence.

```
import re

sentence = "\"Oh, good gracious me!\" said Lucy, suddenly collapsing and again seeing the whole of life in a new perspective."

hits = re.search( r'["](.+)["]' , sentence )

if hits:
    print( hits.group(1) )
## prints Oh, good gracious me!
```

Listing 3.6.

Finding and replacing text

In Python, regular expressions can also be applied usefully in 'find and replace' operations. Such operations can be performed using the sub() function from the 're' module. The sub() function demands three parameters: a regular expression, a replacement string, and the string containing text which needs to be replaced. If matches can be found for the regular expression which is mentioned as the first parameter, these matches will all be replaced with the string which is given as the second parameter.

```
import re

sentence = "This,, ..sentence-. .,contains. .strange. !=puncuation"

sentence = re.sub( r'[.,!=-]' , '' , sentence )

print(sentence)
## This code removes all punctuation
```

Listing 3.7.

Reading the contents of a folder

Part 2 of this tutorial contained an explanation of how individual texts could be opened. Text and data mining projects generally consist of analyses of a large numbers of texts. If all the texts that form a project's corpus are strored in a single folder, the full contents of this folder can be read by making use of the module 'os' ('operating system'). This module has a method named listdir() which, as is suggested by the name, lists all the files in the directory. The result is an object which represents the entire folder. The individual files can subsequently be accessed via a loop which you initiate via the 'for' keyword, as follows:

```
import os

dir = "TXT"

for filename in os.listdir(dir):
    print ( filename )

```

Listing 3.8.