**PYTHON INTRODUCTION TO REGULAR EXPRESSIONS WEEK 2 DAY 4**

**Introduction - Regular expressions**

Regular expressions are a compact way of representing a collection of strings. The power of regular expression is shown in that a single regular expression can represent an unlimited number of strings, but only if the expression’s requirements are met. Regular expressions (also known as ‘regexes’) are defined by using a language other than Python (a mini-language). The way Python communicates and creates expressions is done with the re module.

Regexes are used for five main reasons:

·         **Parsing**: Identifying and extracting pieces of text that match certain criteria.

·         **Searching**: Locating substrings that can have more than one form like: ‘**pet.png**’, ‘**pet.gif**’, ‘**pet.mpg**’ while avoiding ‘**carpet.gif**’.

·         **Searching and replacing**: Find substrings and replace specified words within the matched string or strings, e.g**.** replacing 071 234 5678 with +2771 234 5678.

·         **Splitting strings**: Splitting a string where a certain character occurs, for example, split comma delimits strings every time a ‘,’ is found.

·         **Validation**: Checking whether a string meets criteria, for example, to check whether an email address is in the standard format.

Regexes are used to create parsers, but they do have limitations:

·         They are only able to deal with recursive (repeating) structured text if the maximum number of recursions is known.

·         Large complex regexes are difficult to maintain.

This is why, when parsing, a tool designed for this purpose is used. For example, use an XML parser for XML. At its simplest an expression is just a character, which can be followed by a quantifier. More complex expressions can include any number of quantified expressions.

**Regular expression concepts and syntax**

In this section we look at the regular expression language in four subsections. The first subsection shows you how to match individual characters or groups of characters. The second subsection shows you how to quantify matches, for example, to match a string once or as many times as possible. Next we cover how to create and group sub-expressions. Lastly you will gain an understanding of how to use language assertions and flags.

The following table lists all the re module’s functions, and a short description:

**Table 1 – re module functions**

|  |  |
| --- | --- |
| **Function syntax** | **Description** |
| compile(pattern, flags=0) | Escape all non-alphanumeric characters in pattern. |
| findall(pattern, string, flags=0) | Return a list of all non-overlapping matches in the string. |
| finditer(pattern, string, flags=0) | Return an iterator over all non-overlapping matches in the string. For each match, the iterator returns a match object. |
| match(pattern, string, flags=0) | Try to apply the pattern at the start of the string, returning a match object, or None if no match was found. |
| purge() | Clear the regular expression cache. |
| search(pattern, string, flags=0) | Scan through a string looking for a match to the pattern, returning a match object, or None if no match was found. |
| split(pattern, string, maxsplit=0, flags=0) | Split the source string by the occurrences of the pattern, returning a list containing the resulting substrings. |
| sub(pattern, repl, string, count=0, flags=0) | Return the string obtained by replacing the leftmost non-overlapping occurrences of the pattern in string by the replacement repl. |
| subn(pattern, repl, string, count=0, flags=0) | Return a 2-tuple containing (new\_string, number). The return value new\_string is the string obtained by replacing the leftmost non-overlapping occurrences of the pattern in the source string by the replacement repl. |
| template(pattern, flags=0) | Compile a template pattern, returning a pattern object |

|  |  |
| --- | --- |
| **NOTE** | The most important re functions will be discussed throughout this section. You should familiarise yourself with all these functions. |
|  |

**Regular expression syntax**

The following list explains what the syntax uses within regular expressions.

**Table 2 – Regular expression syntax**

|  |  |
| --- | --- |
| **Syntax** | **Description** |
| . | (Dot) In its original state, dot matches any character except for a newline character. If the DOTALL flag is used, all the characters are matched. |
| ^ | Matches the start of the string. When using the MULTILINE flag, it also matches immediately after each newline or it can be used as a negating character. |
| $ | Matches the end of a string, the character just before the newline. In **MULTILINE** mode it also matches before a newline. But the difference is, 'man' matches both 'man' and 'manners', while the regular expression 'man$' will only match 'man'. More interestingly, try searching for 'number.$' in 'number1\nnumber2\n' matches 'number2' normally, but 'number1' in MULTILINE mode. |
| \* | Causes re to match zero or more occurrences of the preceding re, for as many repetitions as are possible. xy\* will match 'x', 'xy', or 'x' followed by any number of 'y's.    **Note:** re is the regular expression module used by Python. |
| + | Causes re to match one or more occurrences of the preceding re. For example, xy+ will match x followed by any non-zero number of y's; it will not match 'x', but it will match 'xyyyyyyy' |
| ? | Causes re to match 0 or 1 occurrences of the preceding re. For example, xy? will match either 'x' or 'xy'. |
| {m} | Specifies that exactly m copies of the previous re should be matched. If fewer matches are found, it would cause the entire re not to match. It will not match if there are more than m occurrences. For example, x{7} will match exactly 7 'x' characters, but not 6 or 8 characters. |
| {m,n} | Would cause re to match from m to n occurrences of the preceding re, attempting to match as many occurrences as possible. For example, x{4,6} will match from 4 to 6 'x' characters. Omitting m specifies a **lower bound** of zero, and omitting n specifies an **infinite upper bound**.    As an example, x{5,}y will match ‘xxxxxy ‘or a thousand 'x' characters followed by a y, but not ‘xxxxy’. The comma cannot be omitted or the modifier would be confused with the previously described form. |
| {m,n}? | Causes re to match from m to n repetitions of the preceding re, attempting to match as few occurrences as possible. This is the **non-greedy** version of the previous qualifier. For example, on the 7-character string 'xxxxxxx', x{3,6} will match 5 'x' characters, but x{3,6}? will only match 3 characters. |
| \ | Either escapes special characters (permitting you to match characters like '\*', '?', and so forth), or signals a special sequence; special sequences are discussed below this table. |
| [] | Used to indicate a set of characters. Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a '-'. Special characters are not active inside sets. For example, [wxy$] will match any of the characters 'w', 'x', 'y', or '$'. [A-Z] will match any uppercase letter, and[a-zA-Z0-9] matches any **letter** or **digit**. Character classes such as \w or \S (defined in **Table 1.18 –**) are also acceptable inside a range, although the characters they match depend on whether LOCALE or UNICODE mode is in force. If you want to include a '-' or a ']' inside a set, precede it with a backslash, or place it as the first character. The pattern []] will match ']'.    You can match the characters not within a range by complementing the set. This is indicated by including a '^' as the first character of the set. '^' elsewhere will simply match the '^' character. For example, [^6] will match any character except for '6', and [^"] will match any character except '"'.    Note that inside [] the special forms and special characters lose their meanings and only the syntaxes described here are valid. For example, \*, +, (, ), and so on are treated as literals inside [], and back references cannot be used inside []. |
| | | X|Y, where X and Y can be arbitrary, recreates a regular expression that will match either X or Y. An arbitrary number of re can be separated by the '|' in this way. This can be used inside groups as well. As the target string is scanned, re separated by '|' are read from left to right. When one pattern completely matches, that branch is accepted. This means that once X matches, Y will not be tested, even if it would produce a longer overall match. In other words, the '|' operator is never greedy. To match a literal '|', use \|, or enclose it inside a character class, as in [|]. |

If you are not using a raw string (r) to express the pattern, remember that Python also uses the backslash as an escape sequence ( \ ) in string literals; if the escape sequence ( \ ) is not recognised by Python’s parser, the backslash and subsequent character are included in the resulting string. However, if Python recognises the resulting sequence, the backslash should be repeated twice. This is complicated and it is highly recommended that you use raw strings for all expressions.

When raw strings are used with regular expressions, you would not need to include two backslash characters to indicate a single backslash character for regular expressions.

The r in front of a regular expression string changes the string to a raw string. Look at the following examples:

·         **Normal string**'[\\w]+'

·         **Raw string**r'[\w]+'

|  |  |
| --- | --- |
| **NOTE** | \w indicates a word in a string; w without the escape character ( \ ) indicates only the letter 'w'. The next section explains these special characters. |
|  |

The following example illustrates that both ways of writing an expression are valid:

**Example 1 – Raw strings:**

|  |
| --- |
| 1 import re  2  3 number=input("Enter three numbers enclosed in brackets, e.g.(011): ")  4  5 raw\_pat = re.compile(r'$') # Indicates:  6                     # a '(' then three numbers followed by a ')'  7  8 pat = re.compile('\\([0-9]{3}\\)$') # Indicates:  9            # a '(' then three numbers followed by a ')'  10  11 raw\_str = raw\_pat.match(number)  12 string = pat.match(number)  13  14 if ((string != None) and (raw\_str != None)):  15   print("The number is in the correct format")  16 else:  17   print("The number is in the incorrect format") |

**Output:**

|  |
| --- |
| >>>  Enter three numbers enclosed in brackets, e.g.(011):(785)  The number is in the correct format  >>> |

You need to import the re module when using regular expressions (line 1). Lines 5 and 8 indicate exactly the same expression. Line 14 checks that both the expressions match the required format, which then prints a message indicating whether or not a valid input was entered. The following section explains why characters such as brackets need an escape character in front of them.

**Characters and Character Classes**

The simplest form an expression can be in is a single character, such as **5** and **b** (if no quantifier is given, then only one occurrence is matched). For example, **you**as a regex expression has only three expressions (no quantifier is specified, so it will match one **y** followed by one **o** followed by one **u**: this will then capture the following **strings**: your, YouTube, fromMe2you)

There are characters that cannot be used as **literals**, like some **‘**special characters**’.** These characters are defined to be used by regex as a way to tell what should be done with the expression. If the special characters need to be used in a regex, they need to be preceded with a backslash(\).

**Table 3 – Python’s escape characters**

|  |  |
| --- | --- |
| **Syntax** | **Description** |
| \\ | Backslash (\) |
| \b | ASCII Backspace (BS) |
| \n | ASCII Linefeed (LF) |
| \t | ASCII Horizontal Tab (TAB) |
| \v | ASCII Vertical Tab (VT) |
| \' | Single quote (') |
| \" | Double quote (‘) |

The above characters are Python’s special characters. Regex’s special characters are: \. ^ $ ? + \* [] {} (). Python’s escape characters can also be used in regex, e.g., the '\t ' character for a newline. These are **metacharacters**. If you want **metacharacters** to be used as **literals**, they have to be preceded by a \.

For example, to change a regex special character to be used as a normal character with **no special meaning** you would use some of the following in regular expressions:

·         \\

·         \.

·         \^

To change a **Python** special character to be used as a normal character with **no special meaning,** you would just exclude the \ backslash. The following example should clear up any misunderstanding.

**Example 2 – Special characters:**

|  |
| --- |
| 1 import re  2  3 print ("=====================================================")  4 sentence = "\tThis sentence includes:\n\t\tA regex special"  5 sentence += " character: +"  6  7 print (sentence)  8 print ("=====================================================")  9                                                #  \_\_\_\_\_\_\_\_\_\_\_  10 plus = re.compile(r'^.\*[\+]+.\*$',re.DOTALL)    #  | include |  11                                               #  | newline |  12                                               #  |\_\_\_\n\_\_\_\_|  13 Python\_newline = re.compile('^.\*[\n]+.\*$')  14  15 matchP = plus.match(sentence)  16 matchN = Python\_newline.match(sentence)  17  18 if (matchP != None):  19   print("Includes a '+'")  20 else:  21   print("Does not include a '+'")  22  23 if (matchN != None):  24   print("Includes a newline: '\\n'")  25 else:  26   print("Does not include a newline: '\\n'") |

**Output:**

|  |
| --- |
| >>>  =====================================================       This sentence includes:            A regex special character: +  =====================================================  Includes a '+'  Includes a newline: '\n'  >>> |

Line 10 contains a regular expression that will match according to the following rules:

1.    ^          **-**      Matches the start of the text.

2.     .\*         **-**      Matches zero or more characters. (Any character, including the **newline** character)

3.    [\+]+ **-**               Matches one or more + characters.

4.    $          **-**       Matches the end of line.

Line 13 contains a regular expression that will match according to the following rules:

1.    ^          **-**      Matches the start of the text.

2.  .\*          **-**      Matches zero or more characters. (Any character, **except** the **newline** character)

3.    [\n]+      **-**       Matches one or more + characters.

4.    $          **-**       Matches the end of line.

We often need to match different characters, one character or another within text. This can be done using the **character class** (this is not a Python class, it is just the term given by regex for **‘set of characters’**). A character class is an expression, and just like any other expression, unless it is quantified (defined in **1.6.2.3**  Quantifiers). It will only match exactly one character. For example, r[ea]d will match hundred and radar, but not read. It is possible to match a range of characters, e.g. 0 to 9. This can be specified as [0-9], instead of specifying [0123456789]. Negating a regex will exclude characters. To check for all characters except digits: [^0-9]would be used.

Rare **metacharacter** cases would**not require a prefixed**\. Look at the following examples:

·         10-02-91

·         10.02.91

·         10/02/91

They are all dates – the same date in fact. If you want to change the dates to be in a standard format, you would use the following code in Python.

**Example 3 – Change date format:**

|  |
| --- |
| import re    pattern = re.compile(r"[-./]") #notice that – is not prefixed by \    s = '10-02-91, 10/02/91, 10.02.91'    print(pattern.sub( r'/', "Dates changed to the new format: " + s)) |

**Output*:***

|  |
| --- |
| Dates changed to the new format:10/02/91, 10/02/91, 10/02/91 |

There are cases when a dash (-) and other characters are seen as a literal and not a **metacharacter.** This is because it is the first character in a character class ‘[]’.

In 10[-./]02[-./]91, dots are not **metacharacters**, only because they are within a character class (they are always literals in a character class). ., ^, ?, +, \* are also always literals within character classes.

The dashes are normally **metacharacters**, although within a character class they are not if they are the first character after the [ character. If the dash were not first in the character class, it would be mandatory to prefix it with a \.

|  |  |
| --- | --- |
| **NOTE** | Some characters must be prefixed with an escape character ( \ ) within a character class to be used without any special meaning. (Unless directly placed after the ‘[‘ ). |
|  |

\, ', " and ^ must be prefixed with a ( \ ) to be used as a literal, even if they are placed directly after the ‘[‘.

The following examples are all valid regular expressions:

·         r"[.\\/]"

·         r"[[\-?]"

·         r"[\"\-\)]"

·         r'["\-\*]'

|  |  |
| --- | --- |
| **NOTE** | Play around with these examples and more. Character classes are a very important when using regular expressions. |
|  |

The following are shortcuts that can also be used in the Character Class []:

**Table 4 – Shortcuts**

|  |  |
| --- | --- |
| **Symbol** | **Description** |
| \A | Matches only at the start of the string. |
| \d | Matches any Unicode digit (that includes [0–9], and also many other digit characters) with the re.ASCII flag |
| \D | Matches any character that is not a Unicode decimal digit. This is the opposite of \d. If the ASCII flag is used this becomes the equivalent of [^0–9] (but the flag affects the entire regular expression, so in such cases using an explicit [^0–9] may be a better choice). |
| \s | Matches Unicode whitespace characters (that include [\t\n\v] etc., and also many other characters, for example the non-breaking spaces mandated by typography rules in many languages). If the ASCII flag is used, only [\t\n\v] is matched (but the flag affects the entire regular expression, so in such cases using an explicit [\t\n\v] may be a better choice). |
| \S | Matches any character that is not a Unicode whitespace character. This is the opposite of \s. If the ASCII flag is used this becomes the equivalent of [^ \t\n\v] (but the flag affects the entire regular expression, so in such cases using an explicit [^ \t\n\v] may be a better choice). |
| \w | Matches Unicode word characters; this includes most characters that can be part of a word in any language, as well as numbers and the underscore. If the ASCII flag is used, only [a–zA–Z0–9\_] is matched (but the flag affects the entire regular expression, so in such cases using an explicit [a–zA–Z0–9\_] may be a better choice). |
| \W | Matches any character that is not a Unicode word character. This is the opposite of \w. If the ASCII flag is used this becomes the equivalent of [^a-zA-Z0-9\_] (but the flag affects the entire regular expression, so in such cases using an explicit [^a-zA-Z0-9\_] may be a better choice). |
| \Z | Matches only at the end of the string. |

The following example illustrates how these shortcuts can be incorporated in a Python program:

**Example 4 – Shortcuts:**

|  |
| --- |
| 1 import re  2  3 print ("==================================================")  4 sentence = "The quick : very quick : brown fox jumps into a fire"  5 print (sentence)  6 print ("==================================================")  7  8 empty = False  9 count = 0  10  11 pattern = re.compile(r'^[\W]\*[\w]\*[\W]\*')  12  13 while (empty == False):  14   if (sentence == ''):  15     empty = True  16     break  17   else:  18     print("Sentence: " + sentence)  19     sentence = pattern.sub( r'', sentence)  20  21     count += 1  22 print ("Number of words:", count) |

**Output*:***

|  |
| --- |
| >>>  ==================================================  The quick : very quick : brown fox jumps into a fire  ==================================================  Sentence: The quick : very quick : brown fox jumps into a fire  Sentence: quick : very quick : brown fox jumps into a fire  Sentence: very quick : brown fox jumps into a fire  Sentence: quick : brown fox jumps into a fire  Sentence: brown fox jumps into a fire  Sentence: fox jumps into a fire  Sentence: jumps into a fire  Sentence: into a fire  Sentence: a fire  Sentence: fire  Number of words: 10  >>> |

This example uses two shorthand notations (\w, and \W). Line 11 matches text based on the following criteria:

1.         ^         **-**      Matches the start of the text.

2.         [\W]\*     **-**      Matches zero or more characters that **do not** match:

[a-zA-Z0-9\_], includes characters like &, ^, \*, } etc.

3.         [\w]\*     **-**       Matches zero or more characters that match:

[a-zA-Z0-9\_], includes characters like A, H, L, 5 etc.

|  |
| --- |
| **Regular expression:** r'^[\W]\*[\w]\*[\W]\*'    **Sentence:**        'The quick : very quick : brown fox jumps into a fire' |

When the expression is read in sequence, it makes more sense. Regex starts at the first character (^) in sentence, which is 'T' [\W]\* indicates that there can be **zero or more** ( \* ) characters that **aren’t** letters, numbers or underscores, [^a-zA-Z0-9\_]; this means that the first part of the regular expression is valid thus far (^[\W]\*).

The next step is to check for **zero or more** valid letters ([a-zA-Z0-9\_]). 'T', 'h', 'e', are valid for [\w]\*. This means that the second part of the expression is also valid, because 'The' is at the beginning ( ^ ) of sentence.

Now that we have '^[\W]\*[\w]\*'. The last part of the expression checks for **zero or more** characters after 'The' that **are not**letters, numbers or underscores. A **space character** will be found in this case. The expression would return 'The '(notice the space is included). You will find that regular expressions are easy when you analyse expressions.

The while loop replaces each word (and other characters that surround it) with nothing (as seen on line 19). Every time a word is removed, count is increased. If a word is removed, the following word becomes the first word in sentence.

The number of words sentence contains is then printed (line 22).

**Quantifiers**

A quantifier has a form **{m,n}** where m and n are the minimum and maximum number of times the expression must match respectively. For example both **e{1,1}e{1,1}** and**e{2,2}**will match **f**ee**lings**, but will not match **belt.**

It will be tedious, and difficult to read if we had to write a quantifier after each expression; regex provides several **shorthand notations**for this problem:

If **only one** number is supplied it is assumed that the quantifier has a minimum and maximum of the same value, thus **e{4}** is the same as **e{4, 4}** and **e{1,1}e{1,1}** is the same as **e{1}e{1}**.

Having a different minimum or maximum is often convenient. For example, to match travelled and traveled, we could use one of the following travel{1, 2}ed, or travell{0, 1}ed. But another way to write this is travell?ed('?' means {0, 1} or 'l'). This means that the extra 'l' can be included, but it is not necessary.

Other quantifiers look like this when broken up:

·         '+' one or more occurrences.**{1, n}**

·         '\*' zero or more occurrences. **{0, n}**

**Example 5 – Quantifiers:**

|  |
| --- |
| import re    pattern = re.compile('travell?ed')   #could also be travel{1,2}ed    print ('=============================================')  s1 = 'We have travelled the world searching for gold'  print (s1)  s2 = 'We traveled everywhere, but found nothing'  print (s2)  print ('=============================================')    print(pattern.sub('traveled', s1))  print ()  print(pattern.sub('travelled', s2)) |

**Output*:***

|  |
| --- |
| >>>  =============================================  We have travelled the world searching for gold  We traveled everywhere, but found nothing  =============================================  We have traveled the world searching for gold    We travelled everywhere, but found nothing  >>> |