Regular Expressions

References

- Check a regex cheatsheet from the link below or the download version in the same folder. **http://www.cbs.dtu.dk/courses/27610/regular-expressions-cheat-sheet-v2.pdf (http://www.cbs.dtu.dk/courses/27610/regular-expressions-cheat-sheet-v2.pdf)**. Collect 'regular expression examples' from net and compare with the cheat sheet.
- Take a look at the full documentation (https://docs.python.org/3/library/re.html#regular-expression-syntax) if you ever need to look up a particular pattern.
- Check out the nice summary tables at this source (http://www.tutorialspoint.com/python/python_reg_expressions.htm).

Searching for Patterns in Text

One of the most common uses for the re module is for finding patterns in text. Let's do a quick example of using the search method in the re module to find some text:

In [4]:

```
import re

patterns = ['term1', 'term2']

text = 'This is a string with term1, but it does not have the other term.'

for pattern in patterns:
    print('Searching for "%s" in:\n "%s"\n' %(pattern,text))

    if re.search(pattern,text):
        print('Match was found. \n')
    else:
        print('No Match was found.\n')
```

```
Searching for "term1" in:
 "This is a string with term1, but it does not have the other term."

Match was found.

Searching for "term2" in:
 "This is a string with term1, but it does not have the other term."

No Match was found.
```

In [5]:

```
# List of patterns to search for
pattern = 'term1'

# Text to parse
text = 'This is a string with term1, but it does not have the other term.'

match = re.search(pattern,text)

type(match)
```

Out[5]:

```
_sre.SRE_Match
```

This **Match** object returned by the search() method is more than just a Boolean or None, it contains information about the match, including the original input string, the regular expression that was used, and the location of the match. Let's see the methods we can use on the match object:

In [6]:

```
# Show start of match
match.start()
```

Out[6]:

```
22
```

In [7]:

```
# Show end
match.end()
```

Out[7]:

```
27
```

Split with regular expressions

Let's see how we can split with the re syntax. This should look similar to how you used the split() method with strings. So we have two ways of split.

In [8]:

```
# Term to split on
split_term = '@'

phrase = 'What is the domain name of someone with the email: hello@gmail.com'

# Split the phrase
re.split(split_term,phrase)
```

Out[8]:

```
['What is the domain name of someone with the email: hello', 'gmail.com']
```

Note how `re.split()` returns a list with the term to split on removed and the terms in the list are a split up version of the string. Create a couple of more examples for yourself to make sure you understand!

Finding all instances of a pattern

You can use `re.findall()` to find all the instances of a pattern in a string. For example:

In [9]:

```
# Returns a list of all matches
re.findall('match','test phrase match is in middle')
```

Out[9]:

```
['match']
```

re Pattern Syntax

This will be the bulk of this lecture on using re with Python. Regular expressions support a huge variety of patterns beyond just simply finding where a single string occurred.

We can use *metacharacters* along with re to find specific types of patterns.

Since we will be testing multiple re syntax forms, let's create a function that will print out results given a list of various regular expressions and a phrase to parse:

In [11]:

```
def multi_re_find(patterns,phrase):
    '''
    Takes in a list of regex patterns
    Prints a list of all matches
    '''
    for pattern in patterns:
        print('Searching the phrase using the re check: %r' %(pattern))
        print(re.findall(pattern,phrase))
        print('\n')
```

Repetition Syntax

There are five ways to express repetition in a pattern:

1. A pattern followed by the meta-character `*` is repeated zero or more times.
2. Replace the `*` with `+` and the pattern must appear at least once.
3. Using `?` means the pattern appears zero or one time.
4. For a specific number of occurrences, use `{m}` after the pattern, where **m** is replaced with the number of times the pattern should repeat.
5. Use `{m,n}` where **m** is the minimum number of repetitions and **n** is the maximum. Leaving out **n** `{m,}` means the value appears at least **m** times, with no maximum.

Comments:

- **Note * and + both are wild cards. However, there is a difference. + indicates at least once.**
- Because `*` has special meanings as above, so usual meaning of `*` elsewhere (e.g. anything) is replaced by '.*'.

- '.' and - are two special characters in Regex. So we need escape them when we really want . or -. For example `[A-Za-z\-\.]+` , the last - and . are escaped and thus they really mean . and - but not special character in regex.
- Distinguish the different usage of {},[],() in regex. For example (a-z) matches a,- and z. A string 'a-z' will match. This is different from [a-z]. Another example, `(\s+|,)` matches spaces or a comma ','.
- The | in `pattern = \s|\d` can be replaced by pattern = `[\s,\d]` .

In [12]:

```
test_phrase = 'sdsd..sssddd...sdddsddd...dsds...dsssss...sdddd'

test_patterns = [ 'sd*',     # s followed by zero or more d's
                'sd+',          # s followed by one or more d's
                'sd?',          # s followed by zero or one d's
                'sd{3}',        # s followed by three d's
                'sd{2,3}',      # s followed by two to three d's
                ]

multi_re_find(test_patterns,test_phrase)
```

```
Searching the phrase using the re check: 'sd*'
['sd', 'sd', 's', 's', 'sddd', 'sddd', 'sddd', 'sd', 's', 's', 's', 's', 's', 's', 'sdddd']


Searching the phrase using the re check: 'sd+'
['sd', 'sd', 'sddd', 'sddd', 'sddd', 'sd', 'sdddd']


Searching the phrase using the re check: 'sd?'
['sd', 'sd', 's', 's', 'sd', 'sd', 'sd', 'sd', 's', 's', 's', 's', 's', 's', 'sd']


Searching the phrase using the re check: 'sd{3}'
['sddd', 'sddd', 'sddd', 'sddd']


Searching the phrase using the re check: 'sd{2,3}'
['sddd', 'sddd', 'sddd', 'sddd']
```

Character Sets

Character sets are used when you wish to match any one of a group of characters at a point in the input. Brackets are used to construct character set inputs. For example: the input `[ab]` searches for occurrences of either **a** or **b**. Let's see some examples:

In [13]:
```
test_phrase = 'sdsd..sssddd...sdddsddd...dsds...dsssss...sdddd'

test_patterns = ['[sd]',    # either s or d
                 's[sd]+']   # s followed by one or more s or d

multi_re_find(test_patterns,test_phrase)
```

```
Searching the phrase using the re check: '[sd]'
['s', 'd', 's', 'd', 's', 's', 's', 'd', 'd', 'd', 's', 'd', 'd', 'd', 's', 'd', 'd', 'd', 'd', 's', 'd', 's',
'd', 's', 's', 's', 's', 's', 's', 'd', 'd', 'd', 'd']


Searching the phrase using the re check: 's[sd]+'
['sdsd', 'sssddd', 'sdddsddd', 'sds', 'sssss', 'sdddd']
```

It makes sense that the first input `[sd]` returns every instance of s or d. Also, the second input `s[sd]+` returns any full strings that begin with an s and continue with s or d characters until another character is reached.

Exclusion

We can use `^` to exclude terms by incorporating it into the bracket syntax notation. For example: `[^...]` will match any single character not in the brackets. **Note ^ indicates 'start of line+' in other place.** Let's see some examples:

In [14]:
```
test_phrase = 'This is a string! But it has punctuation. How can we remove it?'
```

Use `[^!.? ]` to check for matches that are not a !,.,?, or space. Add a `+` to check that the match appears at least once. This basically translates into finding the words.

```
In [15]: re.findall('[^!.? ]+',test_phrase)
```

```
Out[15]: ['This',
          'is',
          'a',
          'string',
          'But',
          'it',
          'has',
          'punctuation',
          'How',
          'can',
          'we',
          'remove',
          'it']
```

Character Ranges

As character sets grow larger, typing every character that should (or should not) match could become very tedious. A more compact format using character ranges lets you define a character set to include all of the contiguous characters between a start and stop point. The format used is `[start-end]` .

Common use cases are to search for a specific range of letters in the alphabet. For instance, `[a-f]` would return matches with any occurrence of letters between a and f.

Let's walk through some examples:

In [16]:

```
test_phrase = 'This is an example sentence. Lets see if we can find some letters.'

test_patterns=['[a-z]+',      # sequences of lower case letters
               '[A-Z]+',      # sequences of upper case letters
               '[a-zA-Z]+',   # sequences of lower or upper case letters
               '[A-Z][a-z]+'] # one upper case letter followed by lower case letters

multi_re_find(test_patterns,test_phrase)
```

```
Searching the phrase using the re check: '[a-z]+'
['his', 'is', 'an', 'example', 'sentence', 'ets', 'see', 'if', 'we', 'can', 'find', 'some', 'letters']


Searching the phrase using the re check: '[A-Z]+'
['T', 'L']


Searching the phrase using the re check: '[a-zA-Z]+'
['This', 'is', 'an', 'example', 'sentence', 'Lets', 'see', 'if', 'we', 'can', 'find', 'some', 'letters']


Searching the phrase using the re check: '[A-Z][a-z]+'
['This', 'Lets']
```

Escape Codes

You can use special escape codes to find specific types of patterns in your data, such as digits, non-digits, whitespace, and more. For example:

Code	Meaning
`\d`	a digit
`\D`	a non-digit
`\s`	whitespace (tab, space, newline, etc.)
`\S`	non-whitespace

Code	Meaning
`\w`	alphanumeric
`\W`	non-alphanumeric

Escapes are indicated by prefixing the character with a backslash `\`. Unfortunately, a backslash must itself be escaped in normal Python strings, and that results in expressions that are difficult to read. Using **raw strings**, created by prefixing the literal value with `r`, eliminates this problem and maintains readability.

Personally, I think this use of `r` to escape a backslash is probably one of the things that block someone who is not familiar with regex in Python from being able to read regex code at first. Hopefully after seeing these examples this syntax will become clear.

The following is from somewhere else. Note: It's important to **prefix your regex patterns with r to ensure that your patterns are interpreted in the way you want them to. Here 'r' means raw strings.**. Else, you may encounter problems to do with escape sequences in strings. For example, "\n" in Python is used to indicate a new line, but if you use the r prefix, it will be interpreted as the raw string "\n" - that is, the character "\" followed by the character "n" - and not as a new line.

The equivalent two ways: For example: `'\\d+'` is equivalent to `r'\d+'`

In [19]:

```
test_phrase = 'This is a string with some numbers 1233 and a symbol #hashtag'

test_patterns=[ '\\d+', # sequence of digits
                '\\D+', # sequence of non-digits
                '\\s+', # sequence of whitespace
                '\\S+', # sequence of non-whitespace
                '\\w+', # alphanumeric characters
                '\\W+', # non-alphanumeric
                ]
multi_re_find(test_patterns,test_phrase)
```

```
Searching the phrase using the re check: '\\d+'
['1233']


Searching the phrase using the re check: '\\D+'
['This is a string with some numbers ', ' and a symbol #hashtag']


Searching the phrase using the re check: '\\s+'
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']


Searching the phrase using the re check: '\\S+'
['This', 'is', 'a', 'string', 'with', 'some', 'numbers', '1233', 'and', 'a', 'symbol', '#hashtag']


Searching the phrase using the re check: '\\w+'
['This', 'is', 'a', 'string', 'with', 'some', 'numbers', '1233', 'and', 'a', 'symbol', 'hashtag']


Searching the phrase using the re check: '\\W+'
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' #']
```

```
In [20]: test_patterns=[ r'\d+', # sequence of digits
                         r'\D+', # sequence of non-digits
                         r'\s+', # sequence of whitespace
                         r'\S+', # sequence of non-whitespace
                         r'\w+', # alphanumeric characters
                         r'\W+', # non-alphanumeric
                         ]

         multi_re_find(test_patterns,test_phrase)
```

```
Searching the phrase using the re check: '\\d+'
['1233']


Searching the phrase using the re check: '\\D+'
['This is a string with some numbers ', ' and a symbol #hashtag']


Searching the phrase using the re check: '\\s+'
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']


Searching the phrase using the re check: '\\S+'
['This', 'is', 'a', 'string', 'with', 'some', 'numbers', '1233', 'and', 'a', 'symbol', '#hashtag']


Searching the phrase using the re check: '\\w+'
['This', 'is', 'a', 'string', 'with', 'some', 'numbers', '1233', 'and', 'a', 'symbol', 'hashtag']


Searching the phrase using the re check: '\\W+'
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' #']
```

Regular expression applications

Creating token patterns in natural language processing

In [ ]:

```
TOKENS_BASIC = '\\S+(?=\\s+)'  #Create the basic token pattern.
# Check cheatsheet: (?=...) is for lookhead assertion. Then check further for example:
# Isaac (?=Asimov) will match 'Isaac ' only if it’s followed by 'Asimov'.

TOKENS_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)' #Create the alphanumeric token pattern

# Given the following string, which of the below patterns is the best tokenizer? If possible,
#you want to retain sentence punctuation as separate tokens, but have '#1' remain a single token.

my_string = "SOLDIER #1: Found them? In Mercea? The coconut's tropical!"

Answer:
r"(\w+|#\d|\?|!)"  #Note the '\w' can replace both number and letters.
```

Creating searching patterns

In [ ]:

```
# Write a regular expression to search for anything in square brackets: pattern1
pattern1 = r"\[.*\]"
# Define a regex pattern to find hashtags: pattern1
pattern1 = r"#\w+"
# Write a pattern that matches both mentions and hashtags
pattern2 = r"([@#]\w+)"
```

In [ ]:

```
([A-Za-z0-9-]+)
# Letters, numbers and hyphens

(\d{1,2}\/\d{1,2}\/\d{4})
# Date (e.g. 21/3/2006)

([^\s]+(?=\.(jpg|gif|png))\.\2)
# jpg, gif or png image

(^[1-9]{1}$|^[1-4]{1}[0-9]{1}$|^50$)
# Any number from 1 to 50 inclusive.
# ^ Start of line +  Note ^ is not always meaning exclusion
# $ End of line +

(#?([A-Fa-f0-9]){3}(([A-Fa-f0-9]){3})?)
# Valid hexadecimal colour code

((?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,15})
# 8 to 15 character string with at least one
# upper case letter, one lower case letter,
# and one digit (useful for passwords).

(\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,6})
# Email addresses

(\<(/?[^\>]+)\>)
# HTML Tags
```

In [ ]:

```
*  0 or more +
*? 0 or more, ungreedy +
+  1 or more +
+? 1 or more, ungreedy +
?  0 or 1 +
?? 0 or 1, ungreedy +
{3}    Exactly 3 +
{3,}   3 or more +
{3,5}  3, 4 or 5 +
{3,5}? 3, 4 or 5, ungreedy +
```

Check link below about greedy and lazy

https://stackoverflow.com/questions/2301285/what-do-lazy-and-greedy-mean-in-the-context-of-regular-expressions (https://stackoverflow.com/questions/2301285/what-do-lazy-and-greedy-mean-in-the-context-of-regular-expressions)