Introduction to Regular Expressions

 (https://github.com/learn-co-curriculum/dsc-introduction-to-regular-expressions) 
(https://github.com/learn-co-curriculum/dsc-introduction-to-regular-expressions/issues/new)

Introduction

In this lesson, we'll learn about how we can use ***Regular Expressions*** for pattern matching and filtering when working with text data.

Objectives

You will be able to:

- Identify common use cases where regular expressions are useful
- Create regex code to capture meaningful patterns found in text

What Are Regular Expressions?

Regular Expressions are a type of pattern that describe some text. We can use these regular expressions to quickly match patterns and filter through text documents. Regular Expressions (or regex, for short) are an important tool anytime we need to pull information from a larger text document without manually reading the entire thing. For data scientists, regex is extremely useful for data gathering. With regex, we can quickly scrape webpages by using regex to search through the html and find the info needed.

Use Cases for NLP

Regex is especially useful for Natural Language Processing. By definition, just about any text document you work with on an NLP task is going to be one that contains a large amount of text. One of the more common NLP-specific use cases for regex is to use regex during the tokenization stage to define the rules for where we should split strings into separate tokens. As an example, NLTK's basic `word_tokenize()` function would split a word that contains an apostrophe into 3 separate tokens -- `'they're'` gets broken into `["they", "'", "re"]`. This is because the word tokenizer has instructions to just grab sequences of letters as the basic tokens, and an apostrophe isn't a letter. When preprocessing text data, it's quite common to use some small regex patterns to create a more intelligent tokenization scheme to avoid problems like this, so that our tokenizer treats words like `'they're'` as a single token.

Creating Basic Patterns

Regex is only as good as the ***Patterns*** we create. We can use these patterns to find, or to replace text. There are many, many things we can do with regex, and covering them all is outside the scope of this lesson. Instead, we'll just focus on some of the more useful, basic patterns that allow us to begin using regex to work with text data.

Let's take a look at a basic regex pattern, to get a feel for what they look like.

```
import re
sentence = 'he said that she said "hello".'
pattern = 'he'
p = re.compile(sentence)
p.findall() # Output will be ['he', 'he, 'he']
```

We define a pattern by a Python string. We can then use the regular expressions library, `re`, to compile this pattern. Once we have a compiled pattern, we just need to pass in a string and the pattern will find every instance of that pattern in the string.

For people new to regex, the results from the pattern above might be surprising at first. The pattern successfully matches the word 'he', but it also matches the letters 'he' that are found inside of the words 'she' and 'hello'. Subsequences inside of larger sequences are fair game to regex. If we just wanted to match the word 'he', we would need to specify that the pattern needs to start and end with a space, or use of ***anchors*** for things like word boundaries.

Ranges, Groups, and Quantifiers

Obviously, we don't want to have to explicitly type every valid match for any search into our pattern. That would defeat the purpose. Luckily, we don't have to type every possible uppercase letter to match on uppercase letters. Instead, we can use a ***Range*** such as `[A-Z]`. This will match any uppercase letter. Ranges are always inside of square brackets. We can put many things inside of ranges at the same time, and regex will match on any of them. For instance, if we wanted to find any uppercase letter, lowercase letter, or digit, we could use `[A-Za-z0-9]`.

Character Classes

Character classes are a special case of ranges. Since it's quite a common task to use ranges to do things like match on words or numbers, regex actually includes character classes as a shortcut. For instance, we could use `\d` to match any digit -- this is equivalent to using `[0-9]`. We could also use `\w` to match on any word. In the same vein, we can use `\D` to get anything that *isn't* a digit, or `\W` to match on everything that isn't a word. There are a few other types of character classes as well. For a full list, check out the cheat sheet below!

Groups and Quantifiers

Groups are kind of like ranges, but they specify an exact pattern to match on. Groups are denoted by parentheses. Whereas `[A-Z0-9]` matches on any uppercase letter or any digit, `(A-Z0-9)` will only match on the sequence `'A-Z0-9'` exactly. This becomes much more useful when paired with ***Quantifiers***, which allows us to specify how many times a group should happen in a row. If we want to specify an exact number of times, we can use curly braces. For instance, a group followed by `{3}` will only match on patterns that have that group repeated exactly 3 times. The most common quantifiers are usually:

- `*` (0 or more times)
- `+` (1 or more times)
- `?` (0 or 1 times)

In this way, we can fill a grouping with any pattern, tell and specify the number of times we can expect to see that pattern. When we include things like ranges, groupings, and quantifiers together, it becomes easy to write a pattern that can match complex things, like email addresses -- take a look at the example provided below, and see if you can figure out how it works!

```
'([A-Za-z]+)@([A-Za-z]+)\.com'
```

This pattern matches basic email addresses like '**joe@gmail.com (mailto:joe@gmail.com)** ', but not '**john.doe@gmail.com (mailto:john.doe@gmail.com)** ', or '**joe@stanford.edu (mailto:joe@stanford.edu)** '. Take a look at the pattern again -- how would you need to modify the pattern in order for it to match either of those, as well?

Always Keep A Cheat Sheet Handy

Regex is confusing, but it gets easier. With that being said, don't worry about trying to memorize all of the different symbols and metacharacters. Instead, focus on how patterns work, and just look up the symbols when you need them. The internet is filled with great regex cheatsheets. Here's an easy one to keep on hand for future reference:

regularexpressions

Anchors

^	Start of string
\A	Start of string
$	End of string
\Z	End of string
\b	Word boundary
\B	Not word boundary
\<	Start of word
\>	End of word

Character Classes

\c	Control character
\s	White space
\S	Not white space
\d	Digit
\D	Not digit
\w	Word
\W	Not word
\x	Hexadecimal digit
\O	Octal digit

POSIX

[:upper:]	Upper case letters
[:lower:]	Lower case letters
[:alpha:]	All letters
[:alnum:]	Digits and letters
[:digit:]	Digits
[:xdigit:]	Hexadecimal digits
[:punct:]	Punctuation
[:blank:]	Space and tab
[:space:]	Blank characters
[:cntrl:]	Control characters
[:graph:]	Printed characters
[:print:]	Printed characters and spaces
[:word:]	Digits, letters and underscore

Assertions

?=	Lookahead assertion
?!	Negative lookahead
?<=	Lookbehind assertion
?!= or ?<!	Negative lookbehind
?>	Once-only Subexpression
?()	Condition [if then]
?()|	Condition [if then else]
?#	Comment

Quantifiers

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

Quantifier Modifiers

"x" below represents a quantifier

x?	Ungreedy version of "x"

Escape Character

\	Escape Character

Metacharacters (must be escaped)

^	[	.
$	{	*
(	\	+
)	|	?
<	>	

Special Characters

\n	New line
\r	Carriage return
\t	Tab
\v	Vertical tab
\f	Form feed
\xxx	Octal character xxx
\xhh	Hex character hh

Groups and Ranges

.	Any character except new line (\n)
(a|b)	a or b
(...)	Group
(?:...)	Passive Group
[abc]	Range (a or b or c)
[^abc]	Not a or b or c
[a-q]	Letter between a and q
[A-Q]	Upper case letter between A and Q
[0-7]	Digit between 0 and 7
\n	nth group/subpattern

Note: Ranges are inclusive.

Pattern Modifiers

g	Global match
i	Case-insensitive
m	Multiple lines
s	Treat string as single line
x	Allow comments and white space in pattern
e	Evaluate replacement
U	Ungreedy pattern

String Replacement (Backreferences)

$n	nth non-passive group
$2	"xyz" in /^(abc(xyz))$/
$1	"xyz" in /^(?:abc)(xyz)$/
$`	Before matched string
$'	After matched string
$+	Last matched string
$&	Entire matched string

Sample Patterns

Pattern	*Will Match*
([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
(#?([A-Fa-f0-9]){3}(([A-Fa-f0-9]){3})?)	Valid hexadecimal colour code
((?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,15})	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

Note: These patterns are intended for reference purposes and have not been extensively tested. Please use with caution and test thoroughly before use.

Summary

In this lesson, we learned about what regular expressions are, how they are used in NLP for specific tasks, and some common patterns and tools in regex.