Regular Expressions - Codealong

Introduction

In this lab, we'll make use of some common regex patterns to search through text.

Objectives

In this lab you will:

- Create regex code to capture meaningful patterns found in text

Getting Started

In this codealong, we're going to explore some of the more common operations used to filter text and match patterns with regular expressions, or *regex* for short.

Using Regex Testers

Working with regex patterns is an iterative process -- it's quite rare that we'll write the pattern that does exactly what we want while capturing all edge cases the first time through. To make this process simpler, most developers and data scientists make use of popular regex tester websites, which allow you to put in a block of text and test your patterns by quickly seeing visually what a pattern will grab out of the text block. During this codealong, we strongly recommend you take some time to use a regex tester website such as Regex Pal (https://www.regexpal.com/) or regexr (https://regexr.com/) so that you can visually inspect how changing your regex pattern affects your results when working towards a correct answer!

Regex Cheat Sheet

When learning regex, the symbols and notation for patterns can seem a bit intimidating. Don't worry about trying to memorize these symbols and patterns -- it's easier to just keep a cheat sheet handy and look them up as needed. Instead, focus on understanding the basic structure of regex patterns, and then look up the syntax needed to build your pattern.

For reference, we've embedded a common Regex Cheat Sheet found online. Core regex syntax tends to be the same across all languages, so you don't have to worry too much about language-specific features for regex, or looking up regex for the wrong language. Although some languages may add in some special bells and whistles for regex, the core functionality and syntax for regex is generally the same across the board.

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.

The Data

We'll be working with a plain text file in this repo called `'menu.txt'`. Our goal will be to use regex to scrape various types of information from this file.

Run the cell below to import everything we'll need, as well as loading in the file itself.

In [1]:
```
import re

with open('menu.txt', 'r') as f:
    file = f.read()

print(file)
```

```
Flatiron School Cafe Menu


Appetizers

Nachos - $10
Calamari - $12
3 Cheese Platter - $8.75

Entrees

Chicken Sandwich - $16.95
A fried chicken sandwich with lettuce, tomato, and mayo. Add cheese for $1.50

Flatiron Steak - $22
A prime cut of Flatiron Steak, cooked to your liking. Comes with a side of vegetables. Add a salad or cup of soup for $3

Garden Salad - $14
A salad with stuff from the garden on our roof. 3 types of dressing available.


Want to place an order for delivery? Call us at (555) 123-8452!
```

Creating a Basic Pattern With Character Classes

We'll start by creating a basic pattern, and then using the `re` library to search through the file for any instances that match this pattern.

The most simple types of patterns we can use are character classes. These allow us to match predefined things such as words or digits. Let's quickly try getting all the digits.

Run the cell below to find all the digits in the document.

In [2]:
```
pattern = '\d'
p = re.compile(pattern)
digits = p.findall(file)
digits
```

Out[2]:
```
['1',
 '0',
 '1',
 '2',
 '3',
 '8',
 '7',
 '5',
 '1',
 '6',
 '9',
 '5',
 '1',
 '5',
 '0',
 '2',
 '2',
 '3',
 '1',
 '4',
 '3',
 '5',
 '5',
 '5',
 '1',
 '2',
 '3',
 '8',
 '4',
 '5',
 '2']
```

It worked, but not in the way we might have expected. This is because the pattern has found every individual number as an individual match. If you look at the order of the numbers and compare it to the menu text above, you notice that they are in the order you'd see if you read them from left to right starting at the top of the menu.

Let's try modifying our pattern, so that it only gets numbers with a dollar sign in front of them.

Escaping Metacharacters

In regex, the dollar sign is a ***Metacharacter***. This character means something. We can't just use the dollar sign in a pattern, anymore that we can use a reserved keyword like `for` as a variable name in Python. Since the dollar sign is a reserved symbol in regex, we'll need to escape it using a `\`. This tells the regex compiler that we are talking about the actual dollar sign symbol, not using it to talk about the end of a string. In the cell below:

- Modify the pattern so that it includes a dollar sign, followed by a digit character class
- Compile the new pattern
- Use the compiled pattern's `.findall()` method on the pattern
- Display the results

NOTE: Make sure there is no space between the dollar sign and the digit character class, since it will treat the space as a character that is a part of the pattern!

In [3]:
```
pattern = '\$\d'
p = re.compile(pattern)
digits = p.findall(file)
digits
```

Out[3]:
```
['$1', '$1', '$8', '$1', '$1', '$2', '$3', '$1']
```

That's better, but still not perfect! Now, the pattern only gets the first digit for prices. For instance, the pattern truncates `$10` to `$1`.

To fix this, we could also get the next 3 characters that follow a match by adding `.{3}` to the end of pattern. Let's try this in the cell below.

In [4]:
```
pattern = '\$\d.{3}'
p = re.compile(pattern)
digits = p.findall(file)
digits
```

Out[4]:
```
['$8.75', '$16.9', '$1.50']
```

This seems to have worked for some, but not all. It also left out any prices have less than 3 characters after the initial match, such as $10.

We need to create a pattern that gets all the numbers in a price. To do this, we'll make use of groups and ranges!

Using Groups, Ranges, and Quantifiers

Groups and ranges allow us to create more complex patterns by grouping them together. In these groups, we can also provide rangers, to make our patterns less verbose without making them any less powerful. If we want to get any lowercase letters, we could create a group that contains a letter range like `[a-z]`. If we wanted to get all letters regardless of case, we could use the range `[a-zA-Z]`. We could even include things like numbers in here. If wanted to create a grouping that matches any numbers 0-9, or any uppercase or lowercase letters between a and z, we could use `[a-zA-z0-9]`.

Run this pattern in the cell below, and see what it matches.

In [5]:
```
pattern = '[a-zA-Z0-9]'
p = re.compile(pattern)
digits = p.findall(file)
digits
```

Out[5]:
```
['F',
 'l',
 'a',
 't',
 'i',
 'r',
 'o',
 'n',
 'S',
 'c',
 'h',
 'o',
 'o',
 'l',
 'C',
 'a',
 'f',
 'e',
 'M',
```

Now, let's use some ranges, groupings, and quantifiers to match on any price in the menu. Write this pattern now in the cell below. Be sure to include the dollar signs in the pattern to match!

NOTE: This one will probably take you a little while to figure out. Take some time to study all the potential cases we want to match inside the menu. Also, don't forget to escape metacharacters such as dollar signs and decimal points!

In [6]:
```
pattern = '(\$\d+\.?\d*)'
p = re.compile(pattern)
digits = p.findall(file)
digits
```

Out[6]:
```
['$10', '$12', '$8.75', '$16.95', '$1.50', '$22', '$3', '$14']
```

Putting It All Together

To end this lab, we'll have you put your newfound regex skills to the test to see if you can write a pattern that gets the phone number from the bottom of the menu. How you match it is up to you, just as long as you get the phone number! It's okay if the dashes and parentheses are included in your match, but not the exclamation point at the end.

In the cell below, write a regex pattern to match the phone number from the menu, and confirm that it works by compiling it and running in it.

In [7]:
```
pattern = '(\(\d{3}\) (\d{3}-\d{4}))'
p = re.compile(pattern)
digits = p.findall(file)
digits
```

Out[7]:
```
[('(555) 123-8452', '123-8452')]
```

Summary

In this lab, we got some practice in using regex to filter text!