Activity 2: Regular Expressions

Instructions:

- Provide explanations for cells which you code in the tutorial by adding a text below the code cell.
- Note: your explanation must include how the code works and libraries used and why they are used.
- Submit **.ipynb and pdf**(do not use Ctrl+p to generate pdf, use some external ipynb to pdf converters Ex:onlineconvertfree.com, vertopal.com.etc) to canvas.
- The similarity score should be less than 15%

Regular Expressions

a language for specifying text search regular expression strings. This practical language is used in every computer language, word processor, and text processing tools like the Unix tools grep or Emacs. Formally, a regular expression is an algebraic notation for characterizing a set of strings. Regular expressions are particularly useful for searching in texts, when we have a pattern to search corpus for and a corpus of texts to search through. A regular expression search function will search through the corpus, returning all texts that match the pattern. The corpus can be a single document or a collection. For example, the Unix command-line tool grep takes a regular expression and returns every line of the input document that matches the expression.

Python includes a builtin module called `re` which provides regular expression matching operations (Click here for the official module documentation). Once the module is imported into your code, you can use all of the available capabilities for performing pattern-based matching or searching using regular expressions.

In [1]:
```
##code block-1
import re
\
def apply_regex(data, pattern):
  for text in data:
    if re.fullmatch(pattern, text):
      print(f"Test string {text} accepted.")
    else:
      print(f"Test string {text} failed!")
```

Let's write a simple regular expression for matching binary strings.

In [2]:
```
##code block-2
# find binary strings
test_strings = ["0", "1", "dog", "hello, world", "123", "00", "10101010111"]
binary_pattern = r'[0-1]+'
apply_regex(test_strings, binary_pattern)
```

```
Test string 0 accepted.
Test string 1 accepted.
Test string dog failed!
Test string hello, world failed!
Test string 123 failed!
Test string 00 accepted.
Test string 10101010111 accepted.
```

In the code block - 2 first a set of test strings are defined to test a simple regular expression. The binary pattern is defined with r' ' to denote it's a regular expression.The format [0-1]+ is for the list of expressions that can be accepted for this case it would be numbers between 0 and 1.

Now, how about for matching 24-bit hexadecimal codes?

In [3]:
```
##code block-3
test_strings = ["#F0F8FF", "#FFF", "#00FFFFF", "#2980BD", "#FAEBD7"]
hexcode_pattern = r'\#[0-9A-F]{5,6}'
apply_regex(test_strings, hexcode_pattern)
```

```
Test string #F0F8FF accepted.
Test string #FFF failed!
Test string #00FFFFF failed!
Test string #2980BD accepted.
Test string #FAEBD7 accepted.
```

The code block -3 implementation follows the same methodology, since the code is trying to match hexadecimal the test strings are defined as such. The pattern to match the required hexadecimal code would be starting with # and all numbers between 0 - 9 and the alphabets between A and F, the length of the expression would be between 5 and 6 characters

Evaluating a `test_string` against a given `re_pattern`.

In [4]:
```
##code block-4

re_pattern = r'(\([0-9][0-9][0-9]\)\s)?[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]'
test_strings = ["00", "999-999-9999", "(111) 111-1111", "(111)111-1111", "989-1830", "241/131/103", "(182).1
apply_regex(test_strings, re_pattern)
```

```
Test string 00 failed!
Test string 999-999-9999 failed!
Test string (111) 111-1111 accepted.
Test string (111)111-1111 failed!
Test string 989-1830 accepted.
Test string 241/131/103 failed!
Test string (182).1903.1021 failed!
Test string (101).101.1001 failed!
Test string +1 382-394-2849 failed!
Test string 324.235.7679 failed!
Test string +1948-374-2947 failed!
Test string 938.844.293 failed!
```

The code block - 4 would test the strings in test strings which meet the requirements of the re_pattern, which would be for (xxx) xxx-xxxx and xxx-xxxx.The function apply_regex is called by passing the test strings and the required RE patterns.

Question 1

Modify the regular expression in the code block -4 to also accept strings that follow the format `xxx.xxx.xxxx` and `+1 xxx-xxx-xxxx` where `x` is a digit between 0 to 9?

Note:Use different Code blocks for each teststring given in the task

In [5]:
```
##your code here
## modified code block-4

re_pattern = r'(\+1\s)?((\([0-9][0-9][0-9]\)\s)|([0-9][0-9][0-9].))?[0-9][0-9][0-9][.\-][0-9][0-9][0-9][0-9]
# re_pattern = r'((\([0-9][0-9][0-9]\)\s)?[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9])|((\+1\s)?[0-9][0-9][0-9].[0-

test_strings = ["00", "999-999-9999", "(111) 111-1111", "(111)111-1111", "989-1830", "241/131/103", "182.190
apply_regex(test_strings, re_pattern)
```

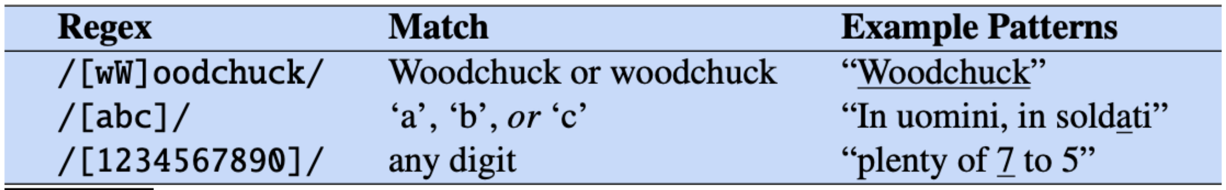
```
Test string 00 failed!
Test string 999-999-9999 accepted.
Test string (111) 111-1111 accepted.
Test string (111)111-1111 failed!
Test string 989-1830 accepted.
Test string 241/131/103 failed!
Test string 182.190.1021 accepted.
Test string +1 190-101-1001 accepted.
```

- The above code tests the string that meets the requirements of the regular expression.
- The above regular expression is for the following patterns.
  - (xxx) xxx-xxxx
  - xxx-xxxx
  - xxx.xxx.xxxx
  - +1 xxx-xxx-xxxx

Basic Regular Expression Patterns

The simplest kind of regular expression is a sequence of simple characters; putting characters in sequence is called concatenation. To search for woodchuck, we type /woodchuck/. The expression /Buttercup/ matches any string containing the substring Buttercup; grep with that expression would return the line I'm called little Buttercup.

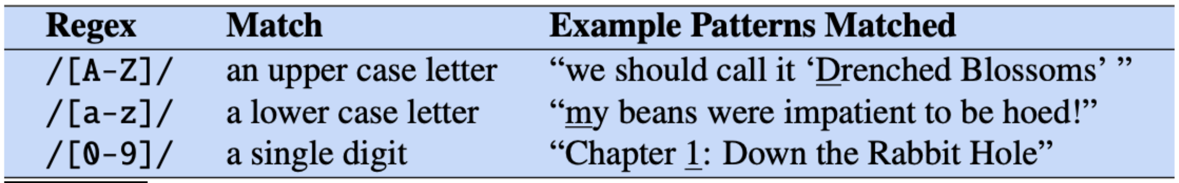
Regular expressions are case sensitive; lower case /s/ is distinct from upper case /S/ (/s/ matches a lower case s but not an upper case S). This means that the pattern /woodchucks/ will not match the string Woodchucks. We can solve this problem with the use of the square braces [and]. The string of characters inside the braces specifies a disjunction of characters to match



Regex	Match	Example Patterns
/[wW]oodchuck/	Woodchuck or woodchuck	"Woodchuck"
/[abc]/	'a', 'b', *or* 'c'	"In uomini, in soldati"
/[1234567890]/	any digit	"plenty of 7 to 5"

Figure 2.2 The use of the brackets [] to specify a disjunction of characters.

The regular expression /[1234567890]/ specifies any single digit. While such classes of characters as digits or letters are important building blocks in expressions, they can get awkward (e.g., it's inconvenient to specify /[ABCDEFGHIJKLMNOPQRSTUVWXYZ]/ to mean "any capital letter"). In cases where there is a well-defined sequence associated with a set of characters, the brackets can be used with the dash (-) to specify any one character in a range. The pattern /[2-5]/ specifies any one of the characters 2, 3, 4, or 5. The pattern /[b-g]/ specifies one of the characters b, c, d, e, f, or g.



Regex	Match	Example Patterns Matched
/[A-Z]/	an upper case letter	"we should call it 'Drenched Blossoms' "
/[a-z]/	a lower case letter	"my beans were impatient to be hoed!"
/[0-9]/	a single digit	"Chapter 1: Down the Rabbit Hole"

Figure 2.3 The use of the brackets [] plus the dash - to specify a range.

The following code block shows a regular expression that matches only those strings that:

1. are at the start of a line and
2. the string does not start with a number or a whitespace

`re.findall()` finds all matches of the pattern in the text under consideration. The output is a list of strings that matched.

In [6]:
```
text = """Here is the First Paragraph and this is the First Sentence. here is the Second Sentence. now is th
Now, it is the Second Paragraph and its First Sentence. here is the Second Sentence."""
```

In [7]:
```
re_pattern1 = r'[A-Z][a-z]+ [A-Z][a-z]+'
print(re.findall(re_pattern1, text))
```

```
['First Paragraph', 'First Sentence', 'Second Sentence', 'Third Sentence', 'Fourth Sentence', 'Fifth Sentenc
e', 'Second Paragraph', 'First Sentence', 'Second Sentence']
```

"Now, let's find occurrences by using regular expressions."

In [8]:
```
##code block-5
import re
text = "cats are cute and dogs are loyal"
pattern = r"\bc[a-z]*|\bd[a-z]*\b"
re.findall(pattern, text)
```

Out[8]:
```
['cats', 'cute', 'dogs']
```

"In the provided code block -5, the given regular expression pattern attempts to identify words that begin with the letters 'c' and 'd'.". \b before c[a-z] *ensures we match full words starting with c |b after d[a-z]* ensures we match full words starting with d

Question 2.

Now try to match words ending in 'ing' or 'ly' by using the given statement

In [9]:
```
##your code here
text = "amazing view lovely painting writing"
pattern = r'\b\w+ing\b|\b\w+ly\b'
re.findall(pattern,text)
```

Out[9]:
```
['amazing', 'lovely', 'painting', 'writing']
```

- The above code is to find all the words that end with either 'ing' or 'ly'
- '\b' is the boundary.
- '\w' means any words and '+' denotes one or more occurances.

Using regular expressions based pattern matching on real world text

For the purposes of demonstration, here's a dummy paragraph of text. A few observations here:

- The text has multiple paragraphs with each paragraph having more than one sentence.
- Some of the words are capitalized (first letter is in uppercase followed by lowercase letters).

In [10]:
```
##code block-6
text = """Here is the First Paragraph and this is the First Sentence. here is the Second Sentence. now is th
Now, it is the Second Paragraph and its First Sentence. here is the Second Sentence. now is the Third Senten
Finally, this is the Third Paragraph and is the First Sentence of this paragraph. here is the Second Sentenc
4th paragraph is not going to be detected by either of the regex patterns below.
"""

print(text)
```

```
Here is the First Paragraph and this is the First Sentence. here is the Second Sentence. now is the Third Se
ntence. this is the Fourth Sentence of the first paragaraph. this paragraph is ending now with a Fifth Sente
nce.
Now, it is the Second Paragraph and its First Sentence. here is the Second Sentence. now is the Third Senten
ce. this is the Fourth Sentence of the second paragraph. this paragraph is ending now with a Fifth Sentence.
Finally, this is the Third Paragraph and is the First Sentence of this paragraph. here is the Second Sentenc
e. now is the Third Sentence. this is the Fourth Sentence of the third paragaraph. this paragraph is ending
now with a Fifth Sentence.
4th paragraph is not going to be detected by either of the regex patterns below.
```

In [11]:
```
re_pattern2 = r'[A-Z][a-z]+ [A-Z][a-z]+'
print(re.findall(re_pattern2, text))
```

```
['First Paragraph', 'First Sentence', 'Second Sentence', 'Third Sentence', 'Fourth Sentence', 'Fifth Sentenc
e', 'Second Paragraph', 'First Sentence', 'Second Sentence', 'Third Sentence', 'Fourth Sentence', 'Fifth Sen
tence', 'Third Paragraph', 'First Sentence', 'Second Sentence', 'Third Sentence', 'Fourth Sentence', 'Fifth
Sentence']
```

The RE pattern is defined to match the sentences for the contigous capitalization of the words.

Following is a text excerpt on "Inaugural Address" taken from the website of the Joint Congressional Committee on Inaugural Ceremonies:

In [12]:
```
inau_text="""The custom of delivering an address on Inauguration Day started with the very first Inauguratio
Every President since Washington has delivered an Inaugural address. While many of the early Presidents read
William Henry Harrison delivered the longest Inaugural address, at 8,445 words, on March 4, 1841—a bitterly
In 1921, Warren G. Harding became the first President to take his oath and deliver his Inaugural address thr
Most Presidents use their Inaugural address to present their vision of America and to set forth their goals
Today, Presidents deliver their Inaugural address on the West Front of the Capitol, but this has not always
```

Question-3

Identify all the capitalized words in the "Inaugural Address" excerpt and write a regular expression that finds all occurrences of such words in the text. Then, run the Python code snippet to automatically display the matched strings according to the pattern.

NOTE: You can use the *re.findall()* method as demonstrated in the example before this exercise.

In [13]:
```
##your code here
re_pattern3 = r'[A-Z][a-z]+'
```

```
print(re.findall(re_pattern3, inau_text))
print("Count = ",len(re.findall(re_pattern3, inau_text)))
```

```
['The', 'Inauguration', 'Day', 'Inauguration', 'George', 'Washington', 'April', 'After', 'Federal', 'Hall',
'New', 'York', 'City', 'Washington', 'Senate', 'Congress', 'His', 'Inauguration', 'Philadelphia', 'March',
'Senate', 'Congress', 'Hall', 'There', 'Washington', 'Inaugural', 'Every', 'President', 'Washington', 'Inaug
ural', 'While', 'Presidents', 'Chief', 'Justice', 'Supreme', 'Court', 'President', 'William', 'Henry', 'Harr
ison', 'Inaugural', 'March', 'He', 'Inauguration', 'Day', 'John', 'Adams', 'Inaugural', 'After', 'Washingto
n', 'Inaugural', 'Franklin', 'Roosevelt', 'January', 'Roosevelt', 'Inauguration', 'White', 'House', 'World',
'War', 'In', 'Warren', 'Harding', 'President', 'Inaugural', 'In', 'Calvin', 'Coolidge', 'Inaugural', 'And',
'Harry', 'Truman', 'President', 'Inaugural', 'Most', 'Presidents', 'Inaugural', 'America', 'Some', 'In', 'Ci
vil', 'War', 'Abraham', 'Lincoln', 'With', 'God', 'In', 'Franklin', 'Roosevelt', 'And', 'John', 'Kennedy',
'And', 'Americans', 'Today', 'Presidents', 'Inaugural', 'West', 'Front', 'Capitol', 'Until', 'Andrew', 'Jack
son', 'Inauguration', 'Presidents', 'House', 'Senate', 'Jackson', 'President', 'East', 'Front', 'Portico',
'Capitol', 'With', 'Inaugurations', 'Ronald', 'Reagan', 'Swearing', 'In', 'Ceremony', 'Inaugural', 'West',
'Front', 'Terrace', 'Capitol', 'The', 'West', 'Front']
Count =  128
```

- The above regular expression finds all the words that have the first letter as capital.
- It prints all the words with the following pattern, which denotes all the words with the first letter capital.
- [A-Z][a-z]+ : This means first letter capital and followed by one or more occurrences of small letters.

Explain briefly the changes you have made in the given Tasks in Today's activity (approximately 200 words)

-->

Your answer here

- For Question-1: In the code block 4, The regular expression or pattern is changed such that the patterns xxx.xxx.xxxx and +1 xxx-xxx-xxxx can also be accepted along with (xxx) xxx-xxxx, xxx-xxxx.
- In the regular expression + denotes that '+' is taken as it is which is known as an escape sequence character. '?' indicates that the pattern before it is optional. [0-9] indicates that it can be any digit. '[.-]' means it can accept either '.' or '-'. '-' has a separate meaning so it is given in an escape sequence character.
- For Question 2, the pattern represents the regular expression that denotes all the words that end with either 'ing' or 'ly'. '\b' indicates the boundaries. '\w' denotes any word. 'text' is the text from which we will find all the words that match the regular expression.
- For Question-3, the regular expression extrans all the words that start with a capital letter followed by small letters. '+' indicates one or more.

Learning:

- Text processing is done first and then followed by natural language processing.
- Regular expressions are used to find a perfect matching pattern.
- Regular expressions are used in generalizing the patterns which can be helpful in classifiers.
- Word tokenizing means splitting the words in a string. That means every word is a token.
- Sentence tokenizing is similar to word tokenizing, but here every sentence is considered as a token.
- There are also various issues in tokenization, such as language issues.

```
In [13]:
```