**Regular Expressions (REGEX)**

martes, 28 de noviembre de 2023

5:14 p. m.

Regular Expressions

Google for Education ref.: <https://developers.google.com/edu/python/regular-expressions?hl=es-419>

Tutorials point ref.: <https://www.tutorialspoint.com/python/python_reg_expressions.htm>

Corey Schafer ref: [Python Tutorial: re Module - How to Write and Match Regular Expressions (Regex)](https://www.youtube.com/watch?v=K8L6KVGG-7o&t=191s)

C. Schafer add. Resource: [Regular Expressions (Regex) Tutorial: How to Match Any Pattern of Text](https://www.youtube.com/watch?v=sa-TUpSx1JA)

Introduction

Regular expressions, or regex, are like magic patterns for searching and manipulating text. They're these sequences of characters that form a search pattern, allowing you to match, find, replace, or validate strings of text based on that pattern.

Imagine you're hunting for specific words in a book, but you don't know the exact words, just a rough idea. Regex is your Sherlock Holmes—it helps you find words by describing their characteristics, like length or specific letters they contain.

regex uses a bunch of symbols and characters to create patterns. Here are some of the basic ones:

* Literals: Regular characters like letters or numbers match themselves. So, the regex "hello" will look for the exact word "hello" in a text.

* Metacharacters: Characters with special meanings. For example, the dot . matches any character, and ^ matches the start of a line.

* Quantifiers: These control how many times a character or group can appear. For instance, \* means "zero or more times," + means "one or more times," and ? means "zero or one time."

* Character classes: Enclosed in square brackets [], they match any one character within the brackets. For instance, [aeiou] matches any vowel.

* Anchors: These specify the position of a match in the text, like ^ for the start of a line and $ for the end.

Regex is super flexible and powerful. You can combine these elements to create complex patterns. Want to match any 10-digit number? You could use '\d{10}' to find exactly that.

A quick example:

Say, it's required to find the first occurrence of a 3-word letter in a string after the string "word:". The following code would do that:

import re

str = 'an example word:cat!!'

match = re.search(r'word:\w\w\w', str)

if match:

    print('found', match.group())   # printout: found word:cat

else:

    print('No occurrences found')

The 're' module

* compile(): The re.compile(*pattern*) function generates a regex compiled object, which it main purpose is to help saving overhead if the same pattern is going to be used repeatedly in the code, because every time a regex function is called, the pattern must be compiled, so to avoid redundant compilation in the processor, this function ease that resources usage. Additionally, by being a regex object, the other functions are callable on this directly.

email\_pattern = re.compile(r'\w+@\w+\.\w+')

text = "Contact us at email@example.com or support@example.org for assistance."

matches = email\_pattern.findall(text)

print(matches)  # ['email@example.com', 'support@example.org']

* search(): The re.search(*pattern*) function catches the first occurrence and returns a regex object.

pattern = re.compile(r'fox')

text = "The quick brown fox jumps over the lazy dog"

match = pattern.search(text)

print(match)  # <re.Match object; span=(16, 19), match='fox'> / A regex object

print(match.group())    # fox / The actual matched string

The groups function works not only to make a readable printout but to separate the groups specified in the pattern with the use of parenthesis:

pattern = re.compile(r'(\w+) (\w+) (\w+)')

text = "The quick brown fox jumps over the lazy fox, while a group of foxes watched"

match = pattern.search(text)

print(match.group())    # The quick brown

print(match.group(1))   # the

print(match.group(2))   # quick

print(match.group(3))   # brown

* match(): works the same as re.search(*pattern*) function with the difference that'll return only if the match is at the very beginning of the string, otherwise it returns false.

* findall(): returns a list with all the occurrences of the pattern found.

text = "The quick brown fox jumps over the lazy dog with another fox"

matches = re.findall(r'fox', text)

print(matches)  # Output: ['fox', 'fox']

* sub(): re.sub( pattern, repl, string, count=0, flags=0) function that replaces all occurrences of the pattern (if not specified otherwise in the count argument) with a replacement string.

text = "The quick brown fox jumps over the lazy dog with another fox"

new\_text = re.sub(r'fox', 'cat', text)

print(new\_text)  # Output: The quick brown cat jumps over the lazy dog with another cat

new\_text = re.sub(r'fox', 'cat', text, count=1)

print(new\_text)  # Output: The quick brown cat jumps over the lazy dog with another fox

* split(): function that splits into substrings according to the pattern passed.

text = "The quick brown fox jumps over the lazy dog"

splitted\_text = re.split(r'o', text)

print(splitted\_text)  # Output: ['The quick br', 'wn f', 'x jumps ', 'ver the lazy d', 'g']

* fullmatch(): This functions work the same as search or match with the difference that is kind of little more strict than the others, because it consider the string in its full extent.

Regex Flags

* **re.IGNORECASE or re.I**: Ignores case when matching. Makes the pattern case-insensitive.

* **re.MULTILINE or re.M**: Allows the '^' and '$' anchors to match the start and end of each line within a multiline string, not just the start and end of the whole string.

* **re.DOTALL or re.S:** Allows the dot '.' in a pattern to match any character, including newline characters ('\n'). Normally, '.' matches any character except newline.

* **re.VERBOSE or re.X**: Allows you to write regular expressions more cleanly by ignoring whitespace and comments within the pattern. This improves readability by allowing you to add comments and spaces.

text = "The quick brown fox\nJumps over the lazy dog"

# Case-insensitive search

pattern = re.compile(r'fox', re.IGNORECASE)

match = pattern.search(text)

if match:

    print("Pattern found:", match.group())  # Pattern found: fox

# Multiline search

pattern = re.compile(r'^Jumps', re.MULTILINE)

match = pattern.search(text)

if match:

    print("Pattern found at the start of a line:", match.group())   # Pattern found at the start of a line: Jumps

# Dotall search

pattern = re.compile(r'brown.\*dog', re.DOTALL)

match = pattern.search(text)

if match:

    print("Pattern found across multiple lines:", match.group())    # Pattern found across multiple lines: brown fox

# Jumps over the lazy dog

# Verbose pattern

pattern = re.compile(r'''

    \b  # Word boundary

    the  # Match 'the'

    \b  # Word boundary

''', re.VERBOSE)

match = pattern.search(text)

if match:

    print("Pattern found using verbose pattern:", match.group())    # Pattern found using verbose pattern: the

Basic Patterns

'a, X, 9, <': The exact match, this means that the regular expression would found any exact match of any character different from ' ^ $ \* + ? { [ ] \ | ( ) ', since those are used for it reserve meaning in patterns.

# without case sensitivity

txt = "Hello there! This is a hello message."

pattern = r'hello'

match = re.findall(pattern, txt, flags=re.IGNORECASE)

if match:

    print('found:', match)   # printout - found: ['Hello', 'hello']

else:

    print('No occurrences found')

# with case sensitivity

txt = "Hello there! This is a hello message."

pattern = r'hello'

match = re.findall(pattern, txt)

if match:

    print('found:', match)   # printout - found: ['hello']

else:

    print('No occurrences found')

The flag re.IGNORECASE switches if the match would be case sensitive or not.

There are some special cases when matching literals: this is the case of a period '.' or a whitespaces or newlines, and for those cases is neccesary to escape them in order to be matched: '\.', '\s', '\n'

# period matching

txt = "Hello there. New message."

pattern = r'\.'

match = re.findall(pattern, txt)

if match:

    print('found:', match)   # printout - found: ['.', '.']

else:

    print('No occurrences found')

It's important to escape the period, otherwise it will mean 'match anything'

# period matching without escaping

txt = "Hello there. New message."

pattern = r'.'

match = re.findall(pattern, txt)

if match:

    print('found:', match)   # printout - found: ['H', 'e', 'l', 'l', 'o', ' ', 't', 'h', 'e', 'r', 'e', '.', ' ', 'N', 'e', 'w', ' ', 'm', 'e', 's', 's', 'a', 'g', 'e', '.']

else:

    print('No occurrences found')

For matching whitespaces happens something similar but with the lower and upper case version of this pattern: with '\s' will match all the whitespaces but if is uppercased '\S' it will 'match anything but the whitespaces'.

# whitespace matching

txt = "Hello there. New message."

pattern = r'\s'

match = re.findall(pattern, txt)

if match:

    print('found:', match)   # printout - found: [' ', ' ', ' ']

else:

    print('No occurrences found')

and

# whitespace matching uppercased

txt = "Hello there. New message."

pattern = r'\S'

match = re.findall(pattern, txt)

if match:

    print('found:', match)   # printout - found: ['H', 'e', 'l', 'l', 'o', 't', 'h', 'e', 'r', 'e', '.', 'N', 'e', 'w', 'm', 'e', 's', 's', 'a', 'g', 'e', '.']

else:

    print('No occurrences found')

More Basic Patterns

'\w': this will match with any letter or digit or underscore, in other words, any character contain in '[zA-Z0-9\_]', and the uppercased version '\W' will match with everything but the characters in that set.

# letter, digits and underscore matching

txt = '''Hello there, again. 15 New messages!'''

pattern = r'\w'

match = re.findall(pattern, txt)

if match:

    print('found:', match)   # printout - found: ['H', 'e', 'l', 'l', 'o', 't', 'h', 'e', 'r', 'e', 'a', 'g', 'a', 'i', 'n', '1', '5', 'N', 'e', 'w', 'm', 'e', 's', 's', 'a', 'g', 'e', 's']

else:

    print('No occurrences found')

and…

# letter, digits and underscore matching

txt = '''Hello there, again. 15 New messages!'''

pattern = r'\W'

match = re.findall(pattern, txt)

if match:

    print('found:', match)   # printout - found: [' ', ',', ' ', '.', ' ', ' ', ' ', '!']

else:

    print('No occurrences found')

'\b': this will match with any 'word boundary' or in other words, will match with a position in the string where a word character (like a letter, digit, or underscore) is followed or preceded by a non-word character (like whitespace, punctuation, or the start/end of the string).

# word boundary

txt = '''Hello there, again. 15 New messages!'''

pattern = r'\bagain\b'

match = re.findall(pattern, txt)

if match:

    print('found:', match)   # printout - found: ['again']

else:

    print('No occurrences found')

'\d': this will match with any digits [0-9].

# digits

txt = '''Hello there, again. 15 New messages!'''

pattern = r'\d'

match = re.findall(pattern, txt)

if match:

    print('found:', match)   # printout - found: ['1', '5']

else:

    print('No occurrences found')

Now, is possible to define the number of digits '\d{n}' being n the number or digits or leaving it open to any-digits string with '\d+'.

# digits

txt = '''Hello there, again. 15 New messages!'''

pattern = r'\d+'

match = re.findall(pattern, txt)

if match:

    print('found:', match)   # printout - found: found: ['15']

else:

    print('No occurrences found')

Metacharacters Patterns

Dot '.' : Matches any character except newline \n. So, a. could match "ab", "ac", "ad", etc. It's like a wildcard for any single character.

txt = '''Hello there, again. 15 New messages!'''

pattern = r'e.'

match = re.findall(pattern, txt)

if match:

    print('found:', match)   # printout - found: ['el', 'er', 'e,', 'ew', 'es', 'es']

else:

    print('No occurrences found')

Caret '^' and Dollar '$' : Represents start and end of a line or string.

pattern\_start = r'^hello'

pattern\_end = r'end$'

text = "hello there, end"

match\_start = re.findall(pattern\_start, text)

match\_end = re.findall(pattern\_end, text)

print(match\_start)  # Output: ['hello']

print(match\_end)    # Output: ['end']

Backslash '\' : Used to escape metacharacters

pattern = r'\.'

text = "A period. Not the end."

matches = re.findall(pattern, text)

print(matches)  # Output: ['.', '.']

Pipe '|' : Acts as an OR operator

pattern = r'cat|dog'

text = "I have a cat and a dog. and you know what? another dog too"

matches = re.findall(pattern, text)

print(matches)  # Output: ['cat', 'dog', 'dog']

Quantifiers

Asterisk '\*': The asterisk \* in regex matches zero or more occurrences of the preceding character or group.

pattern = r'ab\*'  # Matches 'a' followed by zero or more 'b's

text = "ac abb abbb abc"

matches = re.findall(pattern, text)

print(matches)  # Output: ['a', 'abb', 'abbb', 'ab']

pattern = r'\bre\w\*ing\b'

text = "Reading is refreshing. The red car is racing."

matches = re.findall(pattern, text, flags=re.IGNORECASE)

print(matches)  # Output: ['Reading', 'refreshing']

Plus '+' : The plus + quantifier in regex matches one or more occurrences of the preceding character or group. It's similar to the asterisk \*, but it requires at least one occurrence of the character or group.

pattern = r'ab+'  # Matches 'a' followed by one or more 'b's

text = "ac abb abbb abc"

matches = re.findall(pattern, text)

print(matches)  # Output: ['abb', 'abbb', 'ab']

Question mark '?' : denotes an optional character or group. It specifies that the preceding character or group may occur zero or one time, making it optional.

pattern = r'colou?r'  # Matches 'colour' or 'color'

text = "The colour of the car is blue, but color is also acceptable."

matches = re.findall(pattern, text)

print(matches)  # Output: ['colour', 'color']

Curly Braces '{}' : denotes a range of ocurrences of predecing character in which the match will be set, it could be a single parameter of repetition {n}, a inclusive range {n, m} or a minimal count (which means n or higher count) {n,} .

Note: a maximal count {, n} is not supported commonly by regex implementations including python's, but if a count like that is needed, a different pattern could be used r'a{1,2}|a{0}' but undesired empty matches appears in the result additional to the result we are looking for.

pattern = r'a{2}'  # Matches with anything with 2 consecutive (a)s

text = "a aa aaa aaaa aaaaaa abaa ava aara"

matches = re.findall(pattern, text)

print(matches)  # Output: ['aa', 'aa', 'aa', 'aa', 'aa', 'aa', 'aa', 'aa', 'aa']

pattern = r'a{1,3}'  # It also matches with inclusive ranges of (a)s

matches = re.findall(pattern, text)

print(matches)  # Output: ['a', 'aa', 'aaa', 'aaa', 'a', 'aaa', 'aaa', 'a', 'aa', 'a', 'a', 'aa', 'a']

pattern = r'a{3,}'  # and lastly It'll also match with values higher than n {n,}

matches = re.findall(pattern, text)

print(matches)  # Output: ['aaa', 'aaaa', 'aaaaaa']

pattern = r'a{1,2}|a{0}'  # Since maximal count is not suported {,n}, the pattern that will suffice for this is this one, for up to 2 count

matches = re.findall(pattern, text)

print(matches) # Output: ['a', '', 'aa', '', 'aa', 'a', '', 'aa', 'aa', '', 'aa', 'aa', 'aa', '', 'a', '', 'aa', '', 'a', '', 'a', '', 'aa', '', 'a', '']

Character Classes

Character classes are a way of saying in regex to match within a range of characters, and is denoted by the use of squared braces [] and multiple kind of ranges could be formed from this [aeiou], [a-z], [A-Za-z], [0-9], [^0-9] .

# [aeiou]

pattern = r'[aeiou]'  # Matches with any vowel

text = "The Price is $20."

matches = re.findall(pattern, text)

print(matches)  # Output: ['e', 'i', 'e', 'i']

# [A-Z]

pattern = r'[A-Z]'  # Matches with any Uppercased letter

text = "The Price is $20."

matches = re.findall(pattern, text)

print(matches)  # Output: ['T', 'P']

# [A-Za-z]

pattern = r'[A-Za-z]'  # Matches with any Uppercased or lowercased letter

text = "The Price is $20."

matches = re.findall(pattern, text)

print(matches)  # Output: ['T', 'h', 'e', 'P', 'r', 'i', 'c', 'e', 'i', 's']

# [0-9]

pattern = r'[0-9]'  # Matches with any digit

text = "The Price is $20."

matches = re.findall(pattern, text)

print(matches)  # Output: ['2', '0']

# [^0-9]

pattern = r'[^0-9]'  # Matches with anything but a digit

text = "The Price is $20."

matches = re.findall(pattern, text)

print(matches)  # Output: ['T', 'h', 'e', ' ', 'P', 'r', 'i', 'c', 'e', ' ', 'i', 's', ' ', '$', '.']

In groups patterns the syntax '(?<= …)'is called 'Look behind' and '(?= …)'is called 'Look ahead' and means that the pattern has preceding or leading specific characters.