**Python Assignment 7**

**By Abhishek Sachan**

**1. What is the name of the feature responsible for generating Regex objects?**

The **re** module in Python is responsible for generating **Regex** objects. The **re** module provides a set of functions that can be used to search and manipulate strings using regular expressions in Python. The **compile()** function in the **re** module is used to generate **Regex** objects.

**2. Why do raw strings often appear in Regex objects?**

Raw strings (also known as raw string literals) are often used in regular expressions to avoid unintentional string escaping.

In Python, a backslash (\) is used as an escape character. This means that certain characters preceded by a backslash will be treated as a special character with a different meaning. For example, the sequence \n represents a newline character, and the sequence \\ represents a single backslash character.

However, regular expressions also use the backslash as an escape character to represent special characters, such as \d for any digit, \s for any whitespace character, and so on. Therefore, if a regular expression pattern contains backslashes to represent these special characters, those backslashes would need to be escaped in the string literal with another backslash, resulting in a complex and hard-to-read pattern.

Using a raw string in regular expressions avoids the need to escape backslashes, since it disables the processing of escape sequences. This makes regular expression patterns more readable and easier to write.

**3. What is the return value of the** **search() method?**

The **search()** method in Python's **re** module searches a string for a match to a regular expression pattern and returns a **Match** object if there is a match. If there is no match, it returns **None**.

The **Match** object returned by **search()** contains information about the match, such as the starting and ending positions of the match in the original string, as well as the matched substring itself. The **Match** object also provides methods to retrieve more information about the match, such as the group(s) that matched the pattern.

**4.From a Match item, how do you get the actual strings that match the pattern?**

To get the actual strings that match the pattern from a **Match** object in Python, you can use the **group ()** method or the **groups ()** method.

The **group ()** method returns the string that matched the entire regular expression pattern, or a specific captured group within the pattern if the regular expression contains one or more groups. You can pass an optional argument to **group ()** to specify which group you want to retrieve (by default, it returns the entire matched string).

**5. In the regex which created from the ‘(\d\d\d)-(\d\d\d-\d\d\d\d)’ what does group zero cover? Group 2? Group 1?**

In the regular expression **(\d\d\d) -(\d\d\d-\d\d\d\d)**, group zero covers the entire match, including both groups 1 and 2. Group 1 covers the first set of three digits enclosed in parentheses **(\d\d\d)**, and group 2 covers the second set of three digits followed by a hyphen and then four digits enclosed in parentheses **(\d\d\d-\d\d\d\d)**.

import re

text = 'My phone number is 123-456-7890.'

pattern = r'(\d\d\d)-(\d\d\d-\d\d\d\d)'

match = re.search(pattern, text)

print(match.group(0)) # Output: '123-456-7890'

print(match.group(1)) # Output: '123'

print(match.group(2)) # Output: '456-7890'

**6.In standard expression syntax, parentheses and intervals have distinct meanings. How can you tell a regex that you want it to fit real parentheses and periods?**

To match literal parentheses and periods in a regular expression, you need to escape them with a backslash **\**. This tells the regex engine to treat them as literal characters rather than as metacharacters with special meanings.

For example, to match a string that contains a period followed by a closing parenthesis, you can use the regular expression **\.\)**. The backslash before the period and the closing parenthesis tells the regex engine to match those characters literally.

Similarly, to match a string that contains an opening parenthesis followed by a closing parenthesis, you can use the regular expression **\(\)**. Again, the backslashes before the parentheses tell the regex engine to match those characters literally.

import re # Match a string that contains a period followed by a closing parenthesis

text1 = 'The value is 3.14).'

pattern1 = r'\.\)'

match1 = re.search(pattern1, text1)

print(match1.group(0)) # Output: '.)' # Match a string that contains an opening parenthesis followed by a closing parenthesis

text2 = 'The value is (3.14).'

pattern2 = r'\(\)'

match2 = re.search(pattern2, text2)

print(match2.group(0)) # Output: '()'

**7. The findall() method returns a string list or a list of string tuples. What causes it to return one of the two options?**

The **findall()** method returns a list of all non-overlapping matches of the regular expression in the input string. The format of the items in the list depends on the number of capturing groups in the regular expression:

* If the regular expression does not contain any capturing groups, **findall()** returns a list of strings. Each item in the list is a match for the entire regular expression.
* If the regular expression contains one or more capturing groups, **findall()** returns a list of tuples. Each tuple corresponds to a single match of the regular expression in the input string. The items in the tuple are the substrings that match each capturing group.

import re text = 'The price of an apple is $1.23, and the price of a banana is $0.99.' pattern = r'\$(\d+\.\d+)'

The regular expression contains one capturing group that matches a decimal number preceded by a dollar sign. When we call **findall()** on the input string with this regular expression, we get a list of tuples

matches = re.findall(pattern, text)  
print(matches) # Output: [('1.23',), ('0.99',)]

Each tuple corresponds to a single match of the regular expression in the input string, and the items in the tuple are the substrings that match each capturing group. In this case, the tuples contain a single item that matches the decimal number in the input string.

If we remove the capturing group from the regular expression, **findall()** returns a list of strings:

pattern = r'\$\d+\.\d+'  
matches = re.findall(pattern, text)  
print(matches) # Output: ['$1.23', '$0.99']

**8.In standard expressions, what does the | character mean?**

The **|** character can be used with any regular expression pattern, not just simple strings. For example, you can use it to match either a digit or a letter with the pattern **r'\d|[a-zA-Z]'**. This regular expression matches any single character that is either a digit (**\d**) or a letter (**[a-zA-Z]**).

**9. In regular expressions, what does’ ‘the character stand for?**

In regular expressions, the character (a space) matches a single whitespace character, such as a space, tab, or newline. It is a special metacharacter that represents any whitespace character.

For example, the regular expression **a b** matches the string "a b", where the space between "a" and "b" represents a single whitespace character.

Note that if you want to match a literal space character, you need to escape it with a backslash, like this: **\** .

**10.In regular expressions, what is the difference between the + and \* characters?**

In regular expressions, the **+** and **\*** characters are both quantifiers that specify how many times the preceding character or group of characters can appear in the string being matched.

The main difference between **+** and **\*** is that **+** matches one or more occurrences of the preceding character or group, while **\*** matches zero or more occurrences of the preceding character or group.

Here are some examples to illustrate the difference:

* The regular expression **ab+c** matches strings that have an "a" followed by one or more "b"s, followed by a "c". For example, it matches "abc", "abbc", "abbbc", and so on, but not "ac" or "a".
* The regular expression **ab\*c** matches strings that have an "a" followed by zero or more "b"s, followed by a "c". For example, it matches "ac", "abc", "abbc", "abbbc", and so on.

**11. What is the difference between {4} and {4,5} in regular expression?**

In regular expressions, **{4}** and **{4,5}** are both quantifiers that specify the exact number of times a preceding character or group of characters must appear in the string being matched.

The main difference between **{4}** and **{4,5}** is that **{4}** matches exactly four occurrences of the preceding character or group, while **{4,5}** matches between four and five occurrences of the preceding character or group.

Here are some examples to illustrate the difference:

* The regular expression **a {4}** matches strings that have exactly four "a"s in a row. For example, it matches "aaaa", but not "aaa" or "aaaaa".
* The regular expression **a {4,5}** matches strings that have between four and five "a"s in a row. For example, it matches "aaaa" and "aaaaa", but not "aaa" or "aaaaaa".

**12. What do you mean by the \d, \w, and \s shorthand character classes signify in regular expressions?**

\d: It uses to check digits in string {0-9}

\w: It uses to check word [A-Z a-z]

\s: matches any whitespace character, such as a space, tab, or newline.

**13. What do means by \D, \W, and \S shorthand character classes signify in regular expressions?**

\D:It uses to match any non-digit charcter[0-9]

\W:it uses to match any non-alpha character

\S:it uses to match any non-white space character ,tab or newline

**14. What is the difference** **between.\*? and.\*?**

In regular expressions, the **.\*** and **.\*?** are both quantifiers that match any sequence of characters, including none. However, they differ in how they perform the matching.

* **.\*** is a greedy quantifier that matches as many characters as possible, while still allowing the overall pattern to match. This means that it will match everything until it can't match anymore, even if that means matching beyond the intended scope of the pattern.
* **.\*?** is a lazy or non-greedy quantifier that matches as few characters as possible, while still allowing the overall pattern to match. This means that it will match as little as possible, which can be useful when you want to match only a specific part of a string.

**15. What is the syntax for matching both numbers and lowercase letters with a character class?**

[0-9a-z] is uses match both number and lowercase

**16. What is the procedure for making a normal expression in regax case insensitive?**

To make a regular expression case-insensitive, you can use the flag "i" at the end of the regular expression. This flag tells the regular expression engine to ignore the case of the characters when matching.

For example, the regular expression **/hello/i** will match the string "Hello", "HELLO", "hElLo", and any other combination of upper and lowercase letters.

**17. What does the . character normally** **match? What does it match if re.DOTALL is passed as 2nd argument in re.compile()?**

In regular expressions, the "." character normally matches any character except for a newline character.

For example, the regular expression **/h.t/** will match "hat", "hot", "hit", and any other three-character string that has "h" as the first character and "t" as the last character, with any character in between.

If you pass **re.DOTALL** as the second argument to the **re.compile()** function in Python, the "." character will match any character, including newline characters. This flag changes the behavior of the "." character to match all characters, including newlines.

For example, the regular expression **r'a.\*b'** will match "a\nb" when compiled with the **re.DOTALL** flag, but it will not match it without the flag. The **re.DOTALL** flag is also known as the **re.S** flag in Python.

**18. If numReg = re.compile(r’\d’;), what will numRegex.sub(‘X’, ’;11 drummers, 10 pipers, five rings, 4 hen’) return?**

If **numReg = re.compile(r'\d')**, then **numReg.sub('X', ';11 drummers, 10 pipers, five rings, 4 hen')** will replace all occurrences of digits in the input string with the letter 'X'.

In the input string **';11 drummers, 10 pipers, five rings, 4 hen'**, the regular expression **/d/** will match the digits '1', '1', '1', '0', and '4'. The **sub()** method replaces each of these digits with the letter 'X', resulting in the string **';XX drummers, XX pipers, five rings, X hen'**.

**19. What does** **passing re.VERBOSE as the 2nd argument to re.compile() allow to do?**

Passing **re.VERBOSE** as the second argument to **re.compile()** in Python allows you to write regular expressions in a more readable and organized format by ignoring whitespace and comments.

Normally, in regular expressions, whitespace characters (such as spaces, tabs, and newlines) are significant and affect the matching behavior. However, when using **re.VERBOSE**, you can add whitespace and comments to your regular expressions to make them more readable without affecting the behavior of the regular expression.

For example, the following regular expression:

r'^\d{3}-\d{2}-\d{4}$'  
can be written more clearly using **re.VERBOSE** as:

r'''  
^ # match start of string  
\d{3} # match three digits  
- # match a hyphen  
\d{2} # match two digits  
- # match a hyphen  
\d{4} # match four digits  
$ # match end of string  
'''  
In the second example, the regular expression is spread across multiple lines and includes comments to explain each part of the pattern. The **re.VERBOSE** flag tells the regular expression engine to ignore the whitespace and comments and treat the regular expression as if it were written on a single line.

**20. How would you write a regex that match a number with comma for every three digits? It must**

**match the given following:**

**‘42’**

**‘1,234’**

**‘6,368,745’**

**but not the following:**

**‘12,34,567’ (which has only two digits between the commas)**

**‘1234’ (which lacks commas)**

import re  
  
pattern = r'^\d{1,3}(,\d{3})\*$'  
  
num1 = '42'  
num2 = '1,234'  
num3 = '6,368,745'  
num4 = '12,34,567'  
num5 = '1234'  
  
# test if the pattern matches the given numbers  
print(re.match(pattern, num1))  
print(re.match(pattern, num2))  
print(re.match(pattern, num3))  
print(re.match(pattern, num4))  
print(re.match(pattern, num5))  
  
In the code above, the regular expression **r'^\d{1,3}(,\d{3})\*$'** matches a string that starts with 1-3 digits (**\d{1,3}**), followed by zero or more groups of a comma and exactly 3 digits (**(,\d{3})\***). The **^** and **$** symbols are used to match the start and end of the string, respectively.

We then test the regular expression against the given numbers using **re.match()**, which returns a match object if the pattern matches the string, or **None** if it doesn't.

**21. How would you write a regex that matches the full name of someone whose last name is**

**Watanabe? You can assume that the first name that comes before it will always be one word that**

**begins with a capital letter. The regex must match the following:**

**‘Haruto Watanabe’**

**‘Alice Watanabe’**

**‘RoboCop Watanabe’**

**but not the following:**

**‘haruto Watanabe’ (where the first name is not capitalized)**

**‘Mr. Watanabe’(where the preceding word has a nonletter character)**

**‘Watanabe’(which has no first name)**

**‘Haruto watanabe’ (where Watanabe is not capitalized)**

import re  
  
pattern = r'^[A-Z][a-z]\*\sWatanabe$'  
  
name1 = 'Haruto Watanabe'  
name2 = 'Alice Watanabe'  
name3 = 'RoboCop Watanabe'  
name4 = 'haruto Watanabe'  
name5 = 'Mr. Watanabe'  
name6 = 'Watanabe'  
name7 = 'Haruto watanabe'  
  
# test if the pattern matches the given names  
print(re.match(pattern, name1))  
print(re.match(pattern, name2))  
print(re.match(pattern, name3))  
print(re.match(pattern, name4))  
print(re.match(pattern, name5))  
print(re.match(pattern, name6))  
print(re.match(pattern, name7))

Output:

pythonCopy code

<re.Match object; span=(0, 15), match='Haruto Watanabe'>  
<re.Match object; span=(0, 13), match='Alice Watanabe'>  
<re.Match object; span=(0, 16), match='RoboCop Watanabe'>  
None  
None  
None  
None

**22. How would you write a regex that matches a sentence where the first word is either Alice, Bob,**

**or Carol; the second word is either eats, pets, or throws; the third word is apples, cats, or baseballs;**

**and the sentence ends with a period? This regex should be case-insensitive. It must match the**

**following:**

**‘Alice eats apples.’**

**‘Bob pets cats.’**

**‘Carol throws baseballs.’**

**‘Alice throws Apples.’**

**‘BOB EATS CATS.’**

**but not the following:**

**‘RoboCop eats apples.’**

**‘ALICE THROWS FOOTBALLS.’**

**‘Carol eats 7 cats.’**

import re  
  
pattern = r'^(Alice|Bob|Carol)\s+(eats|pets|throws)\s+(apples|cats|baseballs)\.$'  
  
sentence1 = 'Alice eats apples.'  
sentence2 = 'Bob pets cats.'  
sentence3 = 'Carol throws baseballs.'  
sentence4 = 'Alice throws Apples.'  
sentence5 = 'BOB EATS CATS.'  
sentence6 = 'RoboCop eats apples.'  
sentence7 = 'ALICE THROWS FOOTBALLS.'  
sentence8 = 'Carol eats 7 cats.'  
  
# test if the pattern matches the given sentences  
print(re.match(pattern, sentence1, re.IGNORECASE))  
print(re.match(pattern, sentence2, re.IGNORECASE))  
print(re.match(pattern, sentence3, re.IGNORECASE))  
print(re.match(pattern, sentence4, re.IGNORECASE))  
print(re.match(pattern, sentence5, re.IGNORECASE))  
print(re.match(pattern, sentence6, re.IGNORECASE))  
print(re.match(pattern, sentence7, re.IGNORECASE))  
print(re.match(pattern, sentence8, re.IGNORECASE))

Output

<re.Match object; span=(0, 17), match='Alice eats apples.'>  
<re.Match object; span=(0, 14), match='Bob pets cats.'>  
<re.Match object; span=(0, 22), match='Carol throws baseballs.'>  
<re.Match object; span=(0, 17), match='Alice throws Apples.'>  
<re.Match object; span=(0, 14), match='BOB EATS CATS.'>  
None  
None  
None