**Python Assignment 7**

Q1. What is the name of the feature responsible for generating Regex objects?

Ans. The feature in Python responsible for generating Regex objects is the "re" module. It provides support for regular expressions (regex) in Python by enabling the creation of pattern objects using the "compile" function or directly through the use of various regex functions. The resulting Regex objects can be used to search, match, and manipulate strings according to the specified pattern.

Q2. Why do raw strings often appear in Regex objects?

Ans. Raw strings are often used in Regex objects because they allow the backslash character () to be interpreted as a literal character, rather than as an escape character that modifies the meaning of the following character(s). In regular expressions, backslashes are commonly used to escape certain characters with special meaning, such as parentheses or the period (which matches any character).

For example, if we want to create a Regex object to match a string containing a backslash followed by the letter "n", we would need to use a raw string to ensure that the backslash is interpreted as a literal character:

import re

regex\_pattern = r"\\n"

regex\_object = re.compile(regex\_pattern)

In this example, the "r" before the string indicates that it should be treated as a raw string. Without the "r", the backslash would be interpreted as an escape character, resulting in an invalid regex pattern.

Using raw strings in Regex objects is therefore a best practice to avoid errors and ensure that the regex pattern is interpreted as intended.

Q3. What is the return value of the search() method?

Ans. In Python, the search() method is a part of the re (regular expression) module, and it returns a match object if the pattern is found, or None if the pattern is not found. The match object contains information about the match, such as the location of the match within the string, the matched substring, and any groups captured by the pattern.

Here's an example:

import re

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

pattern = r"brown (\w+) jumps"

match = re.search(pattern, text)

if match:

print("Match found:", match.group())

print("Captured group:", match.group(1))

else:

print("Match not found.")

In this example, the search() method is used to search for a pattern that matches the words "brown" and "jumps", with a capture group for the word that comes after "brown". If the pattern is found, the program prints information about the match, including the entire matched string (match.group()), and the captured group (match.group(1)), which in this case is "fox". If the pattern is not found, the program prints "Match not found."

Q4. From a Match item, how do you get the actual strings that match the pattern?

Ans. In Python, you can use the group() method of a match object to get the actual strings that match the pattern. The group() method returns the entire matched string, or a specific captured group if the pattern contains groups.

Here's an example:

import re

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

pattern = r"brown (\w+) jumps"

match = re.search(pattern, text)

if match:

# Get the entire matched string

print("Matched string:", match.group())

# Get the captured group

print("Captured group:", match.group(1))

else:

print("Match not found.")

In this example, the group() method is used to get the entire matched string (match.group()), which is "brown fox jumps". The group(1) method is used to get the captured group (match.group(1)), which is "fox".

If the pattern contains multiple groups, you can use group(n) to get the nth captured group, where n is the group number (starting from 1).

Q5. In the regex which created from the r'(\d\d\d)-(\d\d\d-\d\d\d\d)', what does group zero cover?

Group 2? Group 1?

Ans. The regular expression r'(\d\d\d)-(\d\d\d-\d\d\d\d)' contains two groups, which are enclosed in parentheses.

Group zero (0) refers to the entire match, including all captured groups. In this case, group zero covers the entire phone number in the format of "###-###-####".

Group one (1) refers to the first group of the regular expression, which is (\d\d\d). This group captures the first three digits of the phone number, which are separated from the last seven digits by a hyphen.

Group two (2) refers to the second group of the regular expression, which is (\d\d\d-\d\d\d\d). This group captures the last seven digits of the phone number, which are also separated from the first three digits by a hyphen.

Here's an example of using this regular expression in Python to extract the phone number and its constituent parts:

import re

text = "My phone number is 555-123-4567."

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

match = re.search(pattern, text)

if match:

print("Phone number:", match.group(0))

print("First three digits:", match.group(1))

print("Last seven digits:", match.group(2))

else:

print("Phone number not found.")

In this example, the search() method is used to find the phone number in the string. If the phone number is found, the program prints the entire phone number (match.group(0)), the first three digits (match.group(1)), and the last seven digits (match.group(2)). If the phone number is not found, the program prints "Phone number not found."

Q6. 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?

Ans. To match literal parentheses and periods in a regular expression, you need to "escape" them by placing a backslash (\) before them. This tells the regular expression engine to treat them as literal characters, rather than as special characters with their usual meanings.

For example, to match a literal left parenthesis, you can use the regular expression \(. Similarly, to match a literal right parenthesis, you can use the regular expression \). To match a literal period, you can use the regular expression \..

Here's an example of using escaped parentheses and periods in a regular expression to match a phone number in the format of "(###) ###-####":

import re

text = "My phone number is (555) 123-4567."

pattern = r'\(\d{3}\) \d{3}-\d{4}'

match = re.search(pattern, text)

if match:

print("Phone number:", match.group())

else:

print("Phone number not found.")

In this example, the regular expression \(\d{3}\) \d{3}-\d{4} is used to match a phone number in the format of "(###) ###-####". The escaped parentheses \( and \) match the literal left and right parentheses, and the escaped period \. can be used to match a literal period if needed.

Q7. The findall() method returns a string list or a list of string tuples. What causes it to return one of the two options?

Ans. The findall() method is a regular expression method available in Python's re module, which searches a string for all occurrences of a specified pattern and returns the matched substrings as a list of strings or a list of tuples.

The method returns a list of strings when the regular expression pattern does not contain any capturing groups. In this case, each element in the list represents a substring that matches the pattern.

On the other hand, if the regular expression pattern contains one or more capturing groups (i.e., enclosed in parentheses), findall() returns a list of tuples. Each tuple in the list corresponds to one match, and each element in the tuple represents the matched substring of a capturing group.

Therefore, the output format of findall() depends on whether the regular expression pattern contains capturing groups or not. If there are no capturing groups, it returns a list of strings, and if there are capturing groups, it returns a list of tuples.

Q8. In standard expressions, what does the | character mean?

Ans. In regular expressions, the | character is known as a pipe or vertical bar and is used as a logical OR operator. It is used to match either the expression before or after it.

For example, the regular expression cat|dog matches either the string "cat" or the string "dog". If either "cat" or "dog" appears in the text being searched, the regular expression engine will consider it a match.

Another example, the regular expression gr(a|e)y matches either "gray" or "grey". The pipe separates the two possible spelling variations for the word "gray/grey".

In general, the | character is used to create a logical choice between two or more options in a regular expression pattern.

Q9. In regular expressions, what does the character stand for?

Ans. In regular expressions (regex), characters are used to define the pattern that we want to match in a given string. There are two types of characters used in regex: literal characters and metacharacters.

Literal characters: These are characters that match themselves. For example, the regular expression pattern cat matches the literal characters "c", "a", and "t" in a string.

Metacharacters: These are special characters that have a special meaning in regex. For example, the ^, $, ., \*, +, ?, |, (), [], and {} characters are all metacharacters with special meanings in regex.

The metacharacters are used to define more complex patterns, such as matching a range of characters, matching repetitions of characters, or matching patterns at the beginning or end of a string.

It's important to note that if you want to use a metacharacter as a literal character, you need to escape it by placing a backslash (\) before it. For example, if you want to match a literal period (.) in a string, you need to use the regular expression \. since the period is a metacharacter that matches any character.

Q10 In regular expressions, what is the difference between the + and \* characters?

Ans. In regular expressions, the + and \* characters are quantifiers that specify the number of times a character or group of characters can appear in the string being matched. The main difference between them is as follows:

The + character matches one or more occurrences of the preceding character or group. For example, the regular expression a+ matches one or more "a" characters in a row, such as "a", "aa", "aaa", etc., but not "b" or "ab".

The \* character matches zero or more occurrences of the preceding character or group. For example, the regular expression a\* matches zero or more "a" characters in a row, such as "", "a", "aa", "aaa", etc., and also matches "b" or "ab" since there are zero occurrences of the "a" character.

In other words, the + character requires at least one occurrence of the preceding character or group, while the \* character allows zero or more occurrences.

Both + and \* characters are known as greedy quantifiers because they match as many occurrences of the preceding character or group as possible while still allowing the overall regular expression pattern to match. This behavior can be changed to be non-greedy by adding a? Character after the quantifier, for example, +? Or \*?. This causes the regular expression engine to match the smallest possible number of occurrences of the preceding character or group.

Q11. What is the difference between {4} and {4,5} in regular expression?

Ans. In regular expressions, {4} and {4,5} are both quantifiers that specify the number of times a preceding character or group should be matched. {4} means that the preceding character or group should be matched exactly four times. For example, the regular expression "a{4}" would match the string "aaaa", but not "aa" or "aaaaa". {4,5} means that the preceding character or group should be matched between four and five times. For example, the regular expression "a{4,5}" would match the strings "aaaa" and "aaaaa", but not "aa" or "aaaaaaaa".

So the main difference between {4} and {4,5} is the range of the number of matches that they allow. {4} matches exactly four times, while {4,5} matches between four and five times.

Q12. What do you mean by the \d, \w, and \s shorthand character classes signify in regular expressions?

Ans. In regular expressions, shorthand character classes are predefined character sets that allow you to match certain types of characters without explicitly listing them. The most commonly used shorthand character classes are:

\d: matches any digit character, which is equivalent to [0-9].

\w: matches any word character, which includes alphanumeric characters (A-Z, a-z, 0-9) and underscore (). It is equivalent to [A-Za-z0-9].

\s: matches any whitespace character, which includes spaces, tabs, and line breaks.

For example, if you wanted to match a string that contains any digit character, you could use the shorthand character class \d instead of listing out all the digit characters individually. Similarly, if you wanted to match a string that contains any word character, you could use the shorthand character class \w. If you wanted to match a string that contains any whitespace character, you could use the shorthand character class \s.

Here are some examples of how these shorthand character classes can be used in regular expressions:

The regular expression \d{3} would match any three-digit number.

The regular expression \w+ would match any word that consists of one or more alphanumeric characters or underscores.

The regular expression \shello\s would match any string that contains the word "hello" with optional whitespace characters before and after it.

Q13. What do means by \D, \W, and \S shorthand character classes signify in regular expressions?

Ans. In regular expressions, the uppercase versions of the shorthand character classes \d, \w, and \s are used to match characters that are not in those classes. Here's what they mean:

\D: matches any character that is not a digit, which is equivalent to [^0-9].

\W: matches any character that is not a word character, which includes any character that is not alphanumeric or an underscore. It is equivalent to [^A-Za-z0-9\_].

\S: matches any character that is not a whitespace character.

For example, if you wanted to match a string that contains any character that is not a digit, you could use the shorthand character class \D instead of listing out all the non-digit characters individually. Similarly, if you wanted to match a string that contains any character that is not a word character, you could use the shorthand character class \W. If you wanted to match a string that contains any character that is not a whitespace character, you could use the shorthand character class \S.

Here are some examples of how these shorthand character classes can be used in regular expressions:

The regular expression \D{3} would match any three-character sequence that does not consist of digits.

The regular expression \W+ would match any sequence of one or more characters that are not alphanumeric or an underscore.

The regular expression \Shello\S would match any string that contains the word "hello" with optional non-whitespace characters before and after it.

Q14. What is the difference between .\*? and .\*?

Ans. The difference between .? and . is that the former is a lazy quantifier that matches as few characters as possible, while the latter is a greedy quantifier that matches as many characters as possible.

The dot (.) in both patterns matches any character except line breaks. The asterisk (\*) is a quantifier that means "match zero or more of the preceding character or group".

In the pattern .\*?, the question mark (?) is a lazy quantifier that modifies the behavior of the preceding quantifier to match as few characters as possible to make the overall match succeed. This is often used in cases where you want to match a string between two specific patterns or delimiters, where the shortest possible match is desired.

In contrast, the pattern .\* is a greedy quantifier that matches as many characters as possible to make the overall match succeed. This means that it will match everything in the string, including characters beyond the desired match.

For example, consider the string "The quick brown fox jumps over the lazy dog". The regular expression pattern ".dog" would match the entire string from the beginning to the last occurrence of "dog", whereas the regular expression pattern ".?dog" would match only "The quick brown fox jumps over the lazy ".

In summary, .? is a lazy quantifier that matches as few characters as possible, whereas . is a greedy quantifier that matches as many characters as possible.

Q15. What is the syntax for matching both numbers and lowercase letters with a character class?

Ans. To match both numbers and lowercase letters with a character class, you can use the following syntax:

[0-9a-z]

This character class will match any single character that is either a digit between 0 and 9, or a lowercase letter between a and z. The dash (-) between 0-9 and a-z indicates a range of characters, while the square brackets [] enclose the character class.

Here are some examples of how this character class can be used in regular expressions:

The regular expression [0-9a-z]+ would match any sequence of one or more characters that are either digits or lowercase letters.

The regular expression [0-9a-z]{3} would match any three-character sequence that consists of digits or lowercase letters.

The regular expression ^[0-9a-z]+$ would match any string that consists entirely of digits or lowercase letters, from the beginning (^) to the end ($) of the string.

Q16. What is the procedure for making a normal expression in regex case insensitive?

Ans. To make a regular expression case-insensitive in regex, you can use the flag or modifier "i" after the closing delimiter of the regex pattern. The "i" flag stands for "ignore case" and tells the regex engine to ignore the case of the characters in the pattern.

Here's an example of how to make a regular expression case-insensitive:

/Hello World/i

In this example, the "i" flag is added after the closing delimiter (/) of the regular expression pattern "Hello World". This means that the regex engine will match the pattern regardless of whether the letters are uppercase or lowercase.

Here are some more examples of how to use the "i" flag to make regular expressions case-insensitive:

The regular expression /^[a-z]+$/i would match any string consisting of one or more lowercase letters, regardless of the case of the input string.

The regular expression /\bcat\b/i would match the word "cat" as a separate word in a case-insensitive way, regardless of whether it appears in uppercase or lowercase letters.

The regular expression /[aeiou]/i would match any vowel in a case-insensitive way, regardless of the case of the input string.

In summary, to make a regular expression case-insensitive in regex, you can add the "i" flag after the closing delimiter of the regular expression pattern.

Q17. What does the . character normally match? What does it match if re.DOTALL is passed as 2nd argument in re.compile()?

Ans. In regular expressions, the dot (.) character normally matches any character except for newline characters.

However, if re.DOTALL (or re.S) is passed as the second argument to re.compile(), then the dot character will match any character, including newline characters. This flag tells the regex engine to treat the entire string as a single line and not to stop at newline characters.

Here's an example of how to use re.compile() with the re.DOTALL flag to match any character, including newline characters:

import re

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

text = 'hello\nworld\n'

match = pattern.match(text)

print(match.group())

In this example, the regular expression pattern .\* matches any character, including newline characters, because the re.DOTALL flag is passed as the second argument to re.compile(). The match.group() method returns the entire string, including the newline characters, because the dot character matches them as well.

In summary, the dot (.) character normally matches any character except for newline characters. If re.DOTALL is passed as the second argument to re.compile(), then the dot character will match any character, including newline characters.

Q18. If numReg = re.compile(r'\d+'), what will numRegex.sub('X', '11 drummers, 10 pipers, five rings, 4 hen') return?

Ans. If numReg = re.compile(r'\d+') and numRegex.sub('X', '11 drummers, 10 pipers, five rings, 4 hen') is called, it will return the string 'X drummers, X pipers, five rings, X hen'.

Here's how this works:

The regular expression r'\d+' matches one or more digits in a row.

The sub() method of the numRegex object replaces all occurrences of the regular expression with the replacement string 'X'.

In the input string '11 drummers, 10 pipers, five rings, 4 hen', there are four substrings of one or more digits: '11', '10', '4', and '5' (which does not match the regular expression because it includes non-digits).

The sub() method replaces all of these substrings with the replacement string 'X', resulting in the output string 'X drummers, X pipers, five rings, X hen'.

Therefore, the expression numRegex.sub('X', '11 drummers, 10 pipers, five rings, 4 hen') returns the string 'X drummers, X pipers, five rings, X hen'.

Q19. What does passing re.VERBOSE as the 2nd argument to re.compile() allow to do?

Ans. Passing re.VERBOSE as the second argument to re.compile() in Python allows for more readable and maintainable regular expressions by allowing the use of whitespace and comments within the regular expression pattern.

Here's how it works:

Normally, whitespace within a regular expression is treated as part of the pattern and is matched literally. This means that if you want to include a space or newline character in your regular expression, you must use escape characters like \s or \n.

However, when you use re.VERBOSE as the second argument to re.compile(), whitespace and comments are ignored by the regex engine. This allows you to use spaces and newlines for formatting and to add comments to explain the regular expression pattern.

To use comments within a re.VERBOSE regular expression pattern, start the comment with a # character and end it with the end of the line.

Here's an example of a regular expression pattern using re.VERBOSE:

import re

pattern = re.compile(r"""

# Match a valid email address

[\w.-]+ # One or more word characters, dots, or hyphens

@ # The @ symbol

[\w.-]+ # One or more word characters, dots, or hyphens

\. # The dot

[a-zA-Z]{2,} # Two or more letters (the TLD)

""", re.VERBOSE)

text = 'My email is john.doe@example.com'

match = pattern.search(text)

print(match.group())

In this example, the regular expression pattern matches a valid email address. The re.VERBOSE flag is passed as the second argument to re.compile() to allow for whitespace and comments within the pattern. The regular expression pattern is written across multiple lines with comments to make it more readable and easier to understand.

Using re.VERBOSE in this way can make complex regular expressions easier to read and maintain, especially when working on large projects.

Q20. 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)

Ans. The regular expression to match numbers with commas every three digits would be:

^\d{1,3}(,\d{3})\*$

This regex starts by matching one to three digits at the beginning of the string using \d{1,3}. It then allows zero or more occurrences of a comma followed by exactly three digits using (,\d{3})\*. The ^ and $ anchors ensure that the entire string matches the pattern.

Using this regex, the following strings will match:

'42'

'1,234'

'6,368,745'

And the following strings will not match:

'12,34,567' (which has only two digits between the commas)

'1234' (which lacks commas)

Q21. 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)

Ans. The regular expression to match the full name of someone whose last name is Watanabe would be:

^[A-Z][a-z]\*\sWatanabe$

This regex starts by matching a single uppercase letter using [A-Z] at the beginning of the string, followed by any number of lowercase letters using [a-z]\*. The \s matches a single whitespace character, and 'Watanabe' is matched literally. The ^ and $ anchors ensure that the entire string matches the pattern.

Using this regex, the following strings will match:

'Haruto Watanabe'

'Alice Watanabe'

'RoboCop Watanabe'

And the following strings will not match:

'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)

Q22. 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.'

Ans. The regular expression to match a sentence that starts with Alice, Bob, or Carol; the second word is either eats, pets, or throws; the third word is either apples, cats, or baseballs; and the sentence ends with a period would be:

^(Alice|Bob|Carol)\s+(eats|pets|throws)\s+(apples|cats|baseballs)\.$

This regex uses the alternation operator | to match any of the three names at the beginning of the string. The \s+ matches one or more whitespace characters between each word. The second and third words are also matched with the alternation operator. The period at the end of the sentence is matched literally, and the ^ and $ anchors ensure that the entire string matches the pattern.

Using this regex, the following strings will match:

'Alice eats apples.'

'Bob pets cats.'

'Carol throws baseballs.'

'Alice throws Apples.'

'BOB EATS CATS.'

And the following strings will not match:

'RoboCop eats apples.'

'ALICE THROWS FOOTBALLS.'

'Carol eats 7 cats.' (which contains a number)