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

Ans. In **Python**: The re module provides regular expression functionality in Python. To create a Regex object, you typically use the **re.compile()** function.

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

Ans. Raw strings are often used in Regex objects to avoid unnecessary escaping of characters. In many programming languages, including Python, backslashes () are used as escape characters. This means that certain characters have special meaning when preceded by a backslash, such as \n representing a newline character or \t representing a tab character.

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

Ans. The search() method is used in regular expressions to search for a pattern within a string. The return value of the search() method depends on the programming language or framework you are using. In Python's re module, the search() method returns a match object if a match is found. The match object contains information about the first occurrence of the pattern within the string. If no match is found, the method returns None.

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

**Ans.** In Python's re module, you can use the group() method of the Match object to retrieve the matched string. Calling group() with no arguments or passing 0 as an argument returns the entire matched string. You can also use group(n) to get a specific capturing group's matched string, where n is the group index.

**5. 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. Group 0: The entire matched string.

In this case, group 0 represents the entire pattern match, which means it covers the entire string that matches the regex.

Group 1: The first group defined by the first set of parentheses (\d\d\d).

Group 1 captures the three-digit number before the hyphen. It corresponds to the pattern (\d\d\d).

Group 2: The second group defined by the second set of parentheses (\d\d\d-\d\d\d\d).

Group 2 captures the seven-digit number after the hyphen. It corresponds to the pattern (\d\d\d-\d\d\d\d).

**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?**

Ans. To match literal parentheses:

Use \( to match an opening parenthesis (.

Use \[ to match an opening square bracket [.

Use \) to match a closing parenthesis ).

Use \] to match a closing square bracket ].

**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 in regular expressions can return either a list of strings or a list of string tuples depending on the presence of capturing groups in the regex pattern.

When the regex pattern contains capturing groups (defined by parentheses), the findall() method returns a list of string tuples. Each tuple corresponds to a match, and the elements of the tuple represent the captured groups in the order they appear in the pattern. The first element of each tuple represents the entire match (equivalent to group 0).

On the other hand, if the regex pattern does not contain any capturing groups, the findall() method returns a list of strings. Each element in the list represents a complete match of the pattern. code bleow

import re

pattern\_with\_groups = r'(\d+)-(\d+)'

pattern\_without\_groups = r'\d+-\d+'

text = '123-456 789-012'

matches\_with\_groups = re.findall(pattern\_with\_groups, text)

matches\_without\_groups = re.findall(pattern\_without\_groups, text)

print(matches\_with\_groups) # [('123', '456'), ('789', '012')]

print(matches\_without\_groups) # ['123-456', '789-012']

In the example, pattern\_with\_groups contains capturing groups, while pattern\_without\_groups does not. The findall() method returns a list of string tuples matches\_with\_groups for pattern\_with\_groups, where each tuple represents the captured groups. For pattern\_without\_groups, the method returns a list of strings matches\_without\_groups representing the complete matches.

So, the presence or absence of capturing groups in the regex pattern determines whether findall() returns a list of string tuples or a list of strings.

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

Ans. In standard expressions, the | character is used to represent the logical OR operator. It is used to specify a choice between two or more alternatives in a regular expression pattern.For example, the regular expression pattern "cat|dog" matches either the string "cat" or the string "dog". So, if you apply this pattern to the input text, it will match any occurrence of "cat" or "dog" within the text.

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

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

Ans. **The "+" character:**

The "+" quantifier means "one or more" occurrences of the preceding element.

It requires the preceding element to occur at least once but allows for additional occurrences.

For example, the pattern "go+l" matches "gol", "gool", "goooool", and so on, but it does not match "gl" because there is no "o" present.

**The "\*" character:**

The "\*" quantifier means "zero or more" occurrences of the preceding element. It allows the preceding element to occur any number of times, including zero occurrences.For example, the pattern "go\*l" matches "gl", "gol", "gool", "goooool", and even an empty string (i.e., "").

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

Ans. **"{4}" expression:**

The "{4}" expression specifies that the preceding element must occur exactly four times.For example, the pattern "a{4}" matches "aaaa" but does not match "aa" or "aaaaa".

**"{4,5}" expression:**

The "{4,5}" expression specifies that the preceding element must occur between four and five times, inclusive.For example, the pattern "a{4,5}" matches "aaaa" and "aaaaa", but it does not match "aa", "aaa", or "aaaaaa".

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

Ans. **\d**: It represents any digit from 0 to 9. It is equivalent to the character range [0-9]. It matches a single digit character.

**\w**: It represents any alphanumeric character, including digits, uppercase and lowercase letters, and underscores. It is equivalent to the character range [a-zA-Z0-9\_]. It matches a single alphanumeric character or an underscore.

**\s**: It represents any whitespace character, including spaces, tabs, and line breaks. It matches a single whitespace character.

These shorthand character classes provide a convenient way to represent commonly used character ranges in regular expressions. Here are a few examples of how they can be used:

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

Ans. \D: It represents any character that is not a digit. It is equivalent to the negation of \d. It matches a single non-digit character.

\W: It represents any character that is not alphanumeric (letter or digit) or an underscore. It is equivalent to the negation of \w. It matches a single non-alphanumeric character or non-underscore character.

\S: It represents any character that is not a whitespace character. It is equivalent to the negation of \s. It matches a single non-whitespace character.

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

Ans. .**\*? :-** This expression represents a non-greedy or lazy match of zero or more occurrences of any character. The ? character following .\* makes it match as few characters as possible while still allowing the overall pattern to match. It will try to match the shortest possible substring that satisfies the rest of the pattern. **.\* :-**This expression represents a greedy match of zero or more occurrences of any character. It matches as many characters as possible while still allowing the overall pattern to match. It will try to match the longest possible substring that satisfies the rest of the pattern.

Top of Form

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

Ans. To match both numbers and lowercase letters using a character class in regular expressions, you can combine the character ranges for numbers and lowercase letters within square brackets . By combining ranges within the square brackets, the character class [0-9a-z] will match any digit or lowercase letter.

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

Ans. To make a regular expression case-insensitive in regex, you can use the "case-insensitive" flag or modifier, depending on the programming language or regex engine you are using. Here are two common approaches:

**Using the "case-insensitive" flag**: In many regex flavors, you can append the flag "i" to the end of your regex pattern to indicate case insensitivity. This flag tells the regex engine to ignore the distinction between uppercase and lowercase letters when matching.

For example, if you have the regex pattern /hello/i, it will match "hello", "Hello", "HELLO", and any other variation of case.Note that the syntax for enabling the case-insensitive flag can vary slightly depending on the programming language or regex implementation you are using. Some languages may use different flags like "IgnoreCase" or provide an option parameter to enable case insensitivity.

**Using character class with both cases**: Another way to achieve case insensitivity is by using character classes. You can include both uppercase and lowercase versions of a letter within a character class to match either case.

For example, the regex pattern /[Hh][Ee][Ll][Ll][Oo]/ will match "hello", "Hello", "HELLO", and so on.

This approach explicitly specifies all possible combinations of uppercase and lowercase letters, allowing flexibility in case matching.It's important to consult the documentation or reference guide of your specific programming language or regex engine to determine the correct syntax for enabling case insensitivity, as it may vary.

**17. 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 a newline character ("\n"). It matches a single occurrence of any character, including letters, digits, special characters, and symbols.For example, the pattern "**a.b**" would match "aab", "acb", "a@b", and so on, where the dot represents any character.However, if you pass **re.DOTALL** as the second argument to the re.compile() function or use the inline flag (?s) in your pattern, the behavior of the dot character changes. In this case, the dot matches any character, including newline characters ("\n").This is useful when you want the dot to match across multiple lines. Without the re.DOTALL flag or (?s) inline flag, the dot normally doesn't match newline characters, so it stops at each newline encountered.

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

Ans. If numRegex is defined as re.compile(r'\d+'), and the sub() method is applied with the replacement string 'X' and the input string '11 drummers, 10 pipers, five rings, 4 hen', the resulting string will be:

'X drummers, X pipers, five rings, X hen'

The sub() method in the regular expressions module (re) performs substitution by replacing occurrences of the pattern specified in the regex with the provided replacement string.

In this case, the pattern \d+ matches one or more consecutive digits. The sub() method replaces all matches of this pattern in the input string with the replacement string 'X'. As a result, all occurrences of digits are replaced with 'X', while the non-digit characters remain unchanged.

Therefore, '11 drummers, 10 pipers, five rings, 4 hen' becomes 'X drummers, X pipers, five rings, X hen'.

**19. 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 you to write more readable and organized regular expressions by ignoring whitespace and adding comments.

When re.VERBOSE is used, the regular expression pattern can span multiple lines and include comments without affecting its functionality. This can greatly enhance the readability and maintainability of complex regular expressions.

Here's an example to illustrate how re.VERBOSE works:

**import re pattern = re.compile(r''' \d{3} # Match three digits**

**- # Match a hyphen**

**\d{4} # Match four digits**

**''', re.VERBOSE)**

**# Using the pattern to search for matches**

**result = pattern.search('Phone number: 123-4567')**

**if result: print("Match found!") else: print("No match found.")**

In this example, the regular expression pattern \d{3}-\d{4} is written in verbose mode using re.VERBOSE. The pattern is broken into multiple lines, and comments are added to explain each component of the pattern.

**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)**

Ans. To match a number with commas for every three digits, while ensuring that there are no consecutive commas and that the number is not lacking commas, you can use the following regular expression:

***import re pattern = re.compile(r'^\d{1,3}(,\d{3})\*$')***

**Explanation of the pattern:**

**^** and **$** represent the start and end of the string, ensuring that the pattern matches the entire string.

\d{1,3} matches one to three digits at the beginning of the string.

(,\d{3})\* matches zero or more occurrences of a comma followed by exactly three digits.

The combination (,\d{3})\* allows for commas followed by three digits to repeat any number of times (including zero), effectively allowing the pattern to match a number with commas for every three digits.If there are no commas, or if there are consecutive commas, the pattern will not match.

***Examples of matching strings:***

'42' - Matches as there are no commas.

'1,234' - Matches as there is a comma after every three digits.

'6,368,745' - Matches as there is a comma after every three digits.

**Examples of non-matching strings:**

'12,34,567' - Does not match as there are only two digits between the first two commas.

'1234' - Does not match as there are no commas.

You can use the pattern object to search for matches or perform other regex operations.

**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)**

Ans. To match the full name of someone whose last name is "Watanabe" with the assumption that the first name before it will always be one word beginning with a capital letter, you can use the following regular expression:

**import re pattern = re.compile(r'^[A-Z][a-zA-Z]\*\sWatanabe$')**

**Explanation of the pattern:**

^ and $ represent the start and end of the string, ensuring that the pattern matches the entire string.

[A-Z] matches an uppercase letter at the beginning of the string (the first name).

[a-zA-Z]\* matches zero or more lowercase or uppercase letters (the remaining part of the first name).

\s matches a whitespace character (the space between the first and last names).

Watanabe matches the literal string "Watanabe" as the last name.

**Examples of matching strings:**

'Haruto Watanabe' - Matches as the first name starts with a capital letter and is followed by a space and then "Watanabe".

'Alice Watanabe' - Matches as the first name starts with a capital letter and is followed by a space and then "Watanabe".

'RoboCop Watanabe' - Matches as the first name starts with a capital letter and is followed by a space and then "Watanabe".

**Examples of non-matching strings:**

'haruto Watanabe' - Does not match as the first name does not start with a capital letter.

'Mr. Watanabe' - Does not match as the preceding word contains a non-letter character.

'Watanabe' - Does not match as there is no first name before "Watanabe".

'Haruto watanabe' - Does not match as "Watanabe" is not capitalized.

You can use the pattern object to search for matches or perform other regex operations.

**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.'

Ans. To match a sentence that follows the given criteria, where the first word is either "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 (.), you can use the following regular expression with the case-insensitive flag:

***import re pattern = re.compile(r'^(?:Alice|Bob|Carol)\s(?:eats|pets|throws)\s(?:apples|cats|baseballs)\.$', re.IGNORECASE)***

**Explanation of the pattern:**

^ and $ represent the start and end of the string, ensuring that the pattern matches the entire sentence.

(?:Alice|Bob|Carol) matches either "Alice", "Bob", or "Carol" as the first word, using a non-capturing group (?:...).

\s matches a whitespace character (space) between the words.

(?:eats|pets|throws) matches either "eats", "pets", or "throws" as the second word.

(?:apples|cats|baseballs) matches either "apples", "cats", or "baseballs" as the third word.

\. matches a literal period (.), indicating the end of the sentence.

The re.IGNORECASE flag is used to make the pattern case-insensitive, allowing it to match the words regardless of their capitalization.

**Examples of matching sentences:**

'Alice eats apples.' - Matches as the first word is "Alice", the second word is "eats", the third word is "apples", and the sentence ends with a period.

'Bob pets cats.' - Matches as the first word is "Bob", the second word is "pets", the third word is "cats", and the sentence ends with a period.

'Carol throws baseballs.' - Matches as the first word is "Carol", the second word is "throws", the third word is "baseballs", and the sentence ends with a period.

'Alice throws Apples.' - Matches as the pattern is case-insensitive, so it matches regardless of the capitalization of the words.

'BOB EATS CATS.' - Matches as the pattern is case-insensitive.

**Examples of non-matching sentences:**

'RoboCop eats apples.' - Does not match as the first word is not "Alice", "Bob", or "Carol".

'ALICE THROWS FOOTBALLS.' - Does not match as the second word is not "eats", "pets", or "throws".

'Carol eats 7 cats.' - Does not match as the third word is not "apples", "cats", or "baseballs".

'Alice throws Apples' - Does not match as the sentence does not end with a period.

You can use the pattern object to search for matches or perform other regex operations.