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

Ans:

The feature responsible for generating regular expression (regex) objects in Python is the re module. This module provides support for working with regular expressions, including functions for compiling regex patterns into regex objects, searching for matches within strings, and performing various manipulations on text based on regex patterns.

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

Ans:

Raw strings are often used in regular expression (regex) objects because they treat backslashes (\) as literal characters rather than escape characters. In Python strings, backslashes are used as escape characters to represent special characters like newline (\n), tab (\t), or Unicode characters (\uXXXX). However, in regex patterns, backslashes are frequently used to escape metacharacters (characters with special meanings in regex), such as ^, $, |, (, ), [, ], {, }, ., \*, +, ?, and \ itself.

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

Ans:

The search() method in Python's regular expression module (re) returns a match object if a pattern is found in the string, and None otherwise.

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

Ans:

To get the actual strings that match the pattern from a match object in Python, you can use the group() method. If the match object is named match, match.group() returns the entire substring that matched the pattern.

Here's how you can use it:

import re

pattern = r'cat'

text = 'The cat sat on the mat.'

match = re.search(pattern, text)

if match:

matched\_string = match.group()

print('Match found:', matched\_string) # Output: 'cat'

else:

print('No match found')

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:

In the regex pattern r'(\d\d\d)-(\d\d\d-\d\d\d\d)':

Group 0 (or Group 0): The entire matched string.

Group 1: The first set of three digits captured by the pattern (\d\d\d).

Group 2: The second set of three digits followed by a hyphen, and then four digits captured by 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 ( and ) and periods . in a regular expression pattern without interpreting them as special characters with their respective meanings in regular expression syntax, you can use the backslash \ to escape them.

Here's how you can do it:

To match a literal parenthesis, use \( to match ( and \). For example, to match the string (hello), you can use the pattern \(.+\).

To match a literal period, use \.. For example, to match the string example.com, you can use the pattern example\.com.

Here's an example:

import re

pattern = r'\(hello\)'

text = '(hello) world'

match = re.search(pattern, text)

if match:

print('Match found:', match.group()) # Output: '(hello)'

else:

print('No match found')

7. 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 in Python's regular expression module (re) returns either a list of strings or a list of string tuples based on the presence of capturing groups (parentheses) in the regular expression pattern.

Here's how it works:

If the regular expression pattern contains capturing groups (parentheses), findall() returns a list of string tuples. Each tuple corresponds to a match, and each element of the tuple corresponds to a capturing group within the pattern.

If the regular expression pattern does not contain any capturing groups, findall() returns a list of strings. Each element of the list corresponds to a complete match.

Here's an example to illustrate:

import re

# Pattern with capturing groups

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

text = '123-456 789-012'

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

print('Matches with groups:', matches\_with\_groups) # Output: [('123', '456'), ('789', '012')]

# Pattern without capturing groups

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

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

print('Matches without groups:', matches\_without\_groups) # Output: ['123-456', '789-012']

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

Ans:

In regular expressions, the | character, known as the pipe symbol or alternation operator, is used to specify multiple alternatives within a pattern. It functions similarly to the logical OR operator, allowing you to match either one pattern or another.

For example, the regular expression pattern cat|dog will match either the string "cat" or the string "dog".

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

Ans:

In regular expressions, the . (period) character, also known as the dot or wildcard, is a special metacharacter that matches any single character except newline (\n). It represents any character in the input string, except for a newline character.

For example, the regular expression pattern a.c will match any string that starts with 'a', ends with 'c', and has any character in between the 'a' and 'c'.

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

Ans:

In regular expressions, the + and \* characters are quantifiers used to specify the number of occurrences of the preceding character or group in the pattern.

Here's the difference between them:

+ (Plus Quantifier):

Matches one or more occurrences of the preceding character or group in the pattern.

The preceding character or group must appear at least once in the input string for the match to occur.

Example: ab+ will match 'ab', 'abb', 'abbb', etc., but not 'a' since there must be at least one 'b' following the 'a'.

Example: \d+ will match one or more digits in a row.

\* (Asterisk Quantifier):

Matches zero or more occurrences of the preceding character or group in the pattern.

The preceding character or group may appear zero or more times in the input string for the match to occur.

Example: ab\* will match 'a', 'ab', 'abb', 'abbb', etc., including 'a' since 'b' is optional.

Example: \d\* will match zero or more digits in a row.

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

Ans:

In regular expressions, {4} and {4,5} are quantifiers used to specify the exact number of occurrences of the preceding character or group in the pattern.

Here's the difference between them:

{4}:

Matches exactly four occurrences of the preceding character or group in the pattern.

The preceding character or group must appear exactly four times in the input string for the match to occur.

Example: \d{4} will match exactly four consecutive digits in a row.

{4,5}:

Matches between four and five occurrences of the preceding character or group in the pattern.

The preceding character or group can appear four or five times in the input string for the match to occur.

Example: \d{4,5} will match four or five consecutive digits in a row.

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

Ans:

In regular expressions, \d, \w, and \s are shorthand character classes that represent certain sets of characters:

\d: Represents any digit character. It matches any single digit from 0 to 9.

\w: Represents any word character. It matches any alphanumeric character (letters, digits, or underscore \_).

\s: Represents any whitespace character. It matches spaces, tabs, and newline characters.

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

\d+: Matches one or more consecutive digit characters.

\w+: Matches one or more consecutive word characters (letters, digits, or underscore).

\s+: Matches one or more consecutive whitespace characters (spaces, tabs, or newlines).

These shorthand character classes are commonly used in regular expressions to simplify patterns that involve digits, alphanumeric characters, and whitespace.

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

Ans:

In regular expressions, \D, \W, and \S are negated or inverse shorthand character classes. They represent any character that is not included in the corresponding shorthand character class \d, \w, and \s, respectively.

Here's what each of them signifies:

\D: Represents any character that is not a digit. It matches any character except for digits (0-9).

\W: Represents any character that is not a word character. It matches any character except for alphanumeric characters (letters, digits, and underscore \_).

\S: Represents any character that is not a whitespace character. It matches any character except for spaces, tabs, and newline characters.

These negated shorthand character classes are useful for matching specific characters that are not digits, word characters, or whitespace characters in regular expressions.

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

Ans:

The difference between .\*? and .\*? lies in how they behave in regular expressions:

.\*?: This expression is a non-greedy or lazy quantifier. It matches any character (except for a newline character) zero or more times, as few times as possible, expanding as needed. It will match the shortest possible sequence of characters that satisfies the overall pattern.

.\*?: This expression is the same as .\*?. It is also a non-greedy or lazy quantifier and behaves identically to the first one.

Both .\*? and .\*? are commonly used in regular expressions when you want to match any sequence of characters in a non-greedy manner, meaning they will match the smallest possible substring that still satisfies the overall pattern.

15. 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 in a regular expression, you can use the following syntax:

[0-9a-z]

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

Ans:

To make a regular expression case insensitive in Python, you can use the re.IGNORECASE flag or the re.I shorthand when compiling the regular expression pattern. This flag tells the regular expression engine to ignore the case of the characters when matching.

Here's how to use it:

import re

pattern = r'hello'

text = 'Hello, World!'

# Using the re.IGNORECASE flag

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

print(matches) # Output: ['Hello']

# Using the re.I shorthand

pattern\_case\_insensitive = re.compile(pattern, re.I)

matches = pattern\_case\_insensitive.findall(text)

print(matches) # Output: ['Hello']

In both examples, the regular expression pattern r'hello' is compiled with the re.IGNORECASE flag or the re.I shorthand, which makes the pattern match case-insensitively. As a result, the pattern matches both 'hello' and 'Hello' in the input string 'Hello, World!'.

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

Ans:

To make a regular expression case insensitive in Python, you can use the re.IGNORECASE flag or the re.I shorthand when compiling the regular expression pattern. This flag tells the regular expression engine to ignore the case of the characters when matching.

Here's how to use it:

import re

pattern = r'hello'

text = 'Hello, World!'

# Using the re.IGNORECASE flag

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

print(matches) # Output: ['Hello']

# Using the re.I shorthand

pattern\_case\_insensitive = re.compile(pattern, re.I)

matches = pattern\_case\_insensitive.findall(text)

print(matches) # Output: ['Hello']

In both examples, the regular expression pattern r'hello' is compiled with the re.IGNORECASE flag or the re.I shorthand, which makes the pattern match case-insensitively. As a result, the pattern matches both 'hello' and 'Hello' in the input string 'Hello, World!'.

18. 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 we use the sub() method to replace all occurrences of digits in the given string '11 drummers, 10 pipers, five rings, 4 hen' with the letter 'X', it will return a new string with all the digits replaced by 'X'.

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 create more readable and maintainable regular expressions by ignoring whitespace and comments within the pattern.

Here's what re.VERBOSE does:

Ignore Whitespace: Any whitespace characters (spaces, tabs, and newlines) and comments within the regular expression pattern are ignored. This allows you to format your regular expressions more clearly, using indentation and line breaks to improve readability without affecting the pattern's functionality.

Allow Comments: You can include comments within the regular expression pattern using the # character. Comments start with # and continue to the end of the line. These comments are ignored by the regular expression engine.

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:

You can write a regular expression to match a number with commas for every three digits using the following pattern:

import re

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

tests = ['42', '1,234', '6,368,745', '12,34,567', '1234']

for test in tests:

if pattern.match(test):

print(f"'{test}' is a match")

else:

print(f"'{test}' is not a match")

Explanation of the pattern:

^: Asserts the start of the string.

\d{1,3}: Matches one to three digits.

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

$: Asserts the end of the string.

This pattern ensures that the number starts with one to three digits, followed by zero or more groups of a comma and exactly three digits. It allows for numbers with commas for every three digits.

Output:

'42' is a match

'1,234' is a match

'6,368,745' is a match

'12,34,567' is not a match

'1234' is not a match

In the output, the pattern correctly matches the numbers '42', '1,234', and '6,368,745' while rejecting '12,34,567' and '1234'.

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:

You can write a regular expression to match the full name of someone whose last name is "Watanabe" with the following pattern:

import re

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

tests = [

'Haruto Watanabe',

'Alice Watanabe',

'RoboCop Watanabe',

'haruto Watanabe',

'Mr. Watanabe',

'Watanabe',

'Haruto watanabe'

]

for test in tests:

if pattern.match(test):

print(f"'{test}' is a match")

else:

print(f"'{test}' is not a match")

Explanation of the pattern:

^: Asserts the start of the string.

[A-Z]: Matches a capital letter for the first name.

[a-zA-Z]\*: Matches zero or more letters for the first name.

\s: Matches a whitespace character between the first and last names.

Watanabe: Matches the literal string "Watanabe".

$: Asserts the end of the string.

This pattern ensures that the first name starts with a capital letter, followed by zero or more letters, and is followed by a whitespace character and the last name "Watanabe".

Output:

'Haruto Watanabe' is a match

'Alice Watanabe' is a match

'RoboCop Watanabe' is a match

'haruto Watanabe' is not a match

'Mr. Watanabe' is not a match

'Watanabe' is not a match

'Haruto watanabe' is not a match

In the output, the pattern correctly matches the full names with the last name "Watanabe" while rejecting cases that do not meet the specified criteria.

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:

You can write a regular expression to match the described sentence pattern with the following pattern:

import re

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

tests = [

'Alice eats apples.',

'Bob pets cats.',

'Carol throws baseballs.',

'Alice throws Apples.',

'BOB EATS CATS.',

'RoboCop eats apples.',

'ALICE THROWS FOOTBALLS.',

'Carol eats 7 cats.'

]

for test in tests:

if pattern.match(test):

print(f"'{test}' is a match")

else:

print(f"'{test}' is not a match")

Explanation of the pattern:

^: Asserts the start of the string.

(Alice|Bob|Carol): Matches one of the specified first names.

\s+: Matches one or more whitespace characters.

(eats|pets|throws): Matches one of the specified verbs.

(apples|cats|baseballs): Matches one of the specified objects.

\.$: Matches a period at the end of the string.

re.IGNORECASE: Makes the pattern case-insensitive.

This pattern ensures that the sentence starts with one of the specified first names, followed by one of the specified verbs, one of the specified objects, and ends with a period.

Output:

'Alice eats apples.' is a match

'Bob pets cats.' is a match

'Carol throws baseballs.' is a match

'Alice throws Apples.' is a match

'BOB EATS CATS.' is a match

'RoboCop eats apples.' is not a match

'ALICE THROWS FOOTBALLS.' is not a match

'Carol eats 7 cats.' is not a match

In the output, the pattern correctly matches the sentences that meet the specified criteria while rejecting sentences that do not.