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

ANS :- In Python, the feature responsible for generating regular expression objects is called the "re" module. The "re" module provides functions and methods for working with regular expressions, including the ability to compile regular expressions into pattern objects (Regex objects).

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

ANS :- Raw strings are commonly used in regular expressions (Regex) to avoid the need for excessive escaping of backslashes. In regular expressions, backslashes are used as escape characters to give special meaning to certain characters or sequences. However, in Python strings, backslashes are also used as escape characters

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

ANS :- The search() method of a Regex object in Python returns a match object if a match is found, or None if no match is found.

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, you can use the `group()` method. The `group()` method returns the substring that was matched by the pattern.

Here's an example:

```python

import re

pattern = r'\d+' # Matches one or more digits

text = 'I have 123 apples and 456 oranges'

match = re.search(pattern, text)

if match:

matched\_string = match.group()

print('Matched string:', matched\_string)

else:

print('No match found.')

```

In the example, the pattern `r'\d+'` is used to match one or more digits. The `search()` method is called on the text `'I have 123 apples and 456 oranges'`, and the resulting match object is stored in the `match` variable.

The `group()` method is then used to retrieve the actual string that matches the pattern. In this case, the matched string is `'123'`, and it is printed as the output.

Note that if our regular expression pattern contains multiple capturing groups defined with parentheses, we can pass an integer argument to the `group()` method to retrieve the specific matched group. The argument `0` (default) retrieves the entire matched string, while `1`, `2`, and so on, retrieve the individual capturing groups.

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 regular expression pattern r'(\d\d\d)-(\d\d\d-\d\d\d\d)', the groups are defined using parentheses. Each set of parentheses creates a capturing group that captures a specific portion of the matched string. In this specific pattern:

Group 0 (or group(0)) represents the entire matched string. It covers the entire pattern from start to end, including both the three-digit number and the hyphen-separated four-digit number.

Group 1 (or group(1)) represents the first capturing group defined by (\d\d\d). It captures the three-digit number before the hyphen.

Group 2 (or group(2)) represents the second capturing group defined by (\d\d\d-\d\d\d\d). It captures the hyphen-separated four-digit number.

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 :- In regular expressions, parentheses and periods have special meanings. If we want to match literal parentheses or periods in our regular expression pattern, we need to escape them using a backslash (\). The backslash indicates that the following character should be treated as a literal character instead of having its special regex meaning. To match a literal parentheses ( or ), and a literal period ., you can use \(, \), and \. in our regular expression pattern, respectively

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 different results based on the presence or absence of capturing groups in the regular expression pattern.

1. If the pattern contains no capturing groups:

findall() returns a list of strings. Each string in the list represents a complete match of the pattern.

2. If the pattern contains one or more capturing groups :

findall() returns a list of tuples (or a list of lists if nested groups are used). Each tuple (or list) in the list represents a complete match of the pattern. Each element within the tuple (or list) corresponds to a capturing group in the pattern, providing the matched value for that group.

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

ANS :- In regular expressions, the | character is known as the pipe or alternation operator. It is used to specify multiple alternative patterns within a regular expression. The | operator allows you to match any one of the patterns separated by the | symbol.

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

ANS :- In regular expressions, the meaning of the character may vary depending on the context or surrounding pattern. Without a specific character mentioned, it's difficult to provide a concise answer

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

ANS :- In regular expressions, the + and \* characters are both quantifiers used to specify the number of occurrences of the preceding pattern. However, they have different meanings: + (plus): Matches one or more occurrences of the preceding pattern. \* (asterisk): Matches zero or more occurrences of the preceding pattern.

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 pattern. Here's the difference between them:

1. {4}: Matches exactly four occurrences of the preceding pattern. Requires the preceding pattern to repeat exactly four times for a match. Example: The pattern a{4} would match "aaaa" but would not match "aa" or "aaaaa".

2. {4,5}: Matches a range of occurrences of the preceding pattern, from a minimum of four to a maximum of five occurrences. Allows the preceding pattern to repeat either four or five times for a match. Example: The pattern a{4,5} would match "aaaa" and "aaaaa" but would not match "aa" or "aaaaaa"

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

ANS :- shorthand character classes in regular expressions:

\d represents any digit character (0-9).

\w represents any word character (alphanumeric characters and underscore).

\s represents any whitespace character (spaces, tabs, newlines, etc.).

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

ANS :- \D represents any non-digit character.

\W represents any non-word character.

\S represents any non-whitespace character

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

ANS :- The difference between .\*? and .\* in regular expressions lies in their matching behavior:

.\*? is a non-greedy or lazy match. It matches as few characters as possible to satisfy the pattern. It will stop matching as soon as the next part of the pattern can be matched.

.\* is a greedy match. It matches as many characters as possible to satisfy the pattern. It will continue matching until the end of the string or until the next part of the pattern cannot be matched.

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

ANS :- Here's an example usage of syntax in Python:

```python

import re

text = 'abc123xyz456'

pattern = r'[0-9a-z]'

matches = re.findall(pattern, text)

print(matches) # Output: ['a', 'b', 'c', '1', '2', '3', 'x', 'y', 'z']

```

In this example, the pattern `[0-9a-z]` is used with the `re.findall()` function to find all occurrences of digits or lowercase letters in the given text. The resulting matches are `['a', 'b', 'c', '1', '2', '3', 'x', 'y', '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, use the `re.IGNORECASE` or `re.I` flag when compiling the regex pattern. For example:

```python

import re

pattern = re.compile(r'your\_pattern', re.IGNORECASE)

```

By including this flag, the regular expression pattern will match regardless of the case of the letters in the text being searched.

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

ANS :- By default, the . (dot) character in a regular expression matches any character except a newline character (\n). It matches a single character from the input string. However, if the re.DOTALL flag is passed as the second argument when compiling the regular expression using re.compile(), the . character will match any character, including newline characters (\n). The re.DOTALL flag enables the dot to match newline characters as well.

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+')`, the expression `numReg.sub('X', '11 drummers, 10 pipers, five rings, 4 hen')` will return the following string: `'X drummers, X pipers, five rings, X hen'`.

The `sub()` method of a compiled regex pattern replaces all occurrences of the pattern in the input string with the specified replacement. In this case, the pattern `\d+` matches one or more digits.

By using `numReg.sub('X', '11 drummers, 10 pipers, five rings, 4 hen')`, the digits in the input string are replaced with the letter 'X', resulting in the modified string where all numbers are replaced.

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() allows you to write regular expressions with improved readability and structure by ignoring whitespace and adding comments. When using re.VERBOSE, you can include spaces, tabs, and newlines within the regular expression pattern to enhance its readability. Additionally, you can use comments within the pattern by starting them with the # symbol.

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 regex pattern to match numbers with commas for every three digits using the following expression:

```python

import re

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

```

Explanation of the pattern:

- `^` asserts the start of the string.

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

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

- `$` asserts the end of the string.

This pattern will match numbers that have commas separating every three digits, including numbers without any commas.

Example usage:

```python

import re

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

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

for number in numbers:

if pattern.match(number):

print(f"Matched: {number}")

else:

print(f"Not matched: {number}")

```

Output:

```

Matched: 42

Matched: 1,234

Matched: 6,368,745

Not matched: 12,34,567

Not matched: 1234

```

In the example, the numbers that match the pattern are printed as "Matched," while the ones that don't match are printed as "Not matched."

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 :- We can write a regex pattern to match the full name of someone whose last name is Watanabe using the following expression:

```python

import re

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

```

Explanation of the pattern:

- `^` asserts the start of the string.

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

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

- ` ` matches a space.

- `Watanabe` matches the exact string "Watanabe" for the last name.

- `$` asserts the end of the string.

This pattern will match names that have a single capitalized first name followed by a space and the last name "Watanabe."

Example usage:

```python

import re

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

names = ['Haruto Watanabe', 'Alice Watanabe', 'RoboCop Watanabe', 'haruto Watanabe', 'Mr. Watanabe', 'Watanabe', 'Haruto watanabe']

for name in names:

if pattern.match(name):

print(f"Matched: {name}")

else:

print(f"Not matched: {name}")

```

Output:

```

Matched: Haruto Watanabe

Matched: Alice Watanabe

Matched: RoboCop Watanabe

Not matched: haruto Watanabe

Not matched: Mr. Watanabe

Not matched: Watanabe

Not matched: Haruto watanabe

```

In the example, the names that match the pattern are printed as "Matched," while the ones that don't match are printed as "Not matched."

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 :- We can write a regex pattern to match the described sentence using the following expression:

```python

import re

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

```

Explanation of the pattern:

- `^` asserts the start of the string.

- `(Alice|Bob|Carol)` matches one of the specified names (case-insensitive).

- `(eats|pets|throws)` matches one of the specified actions (case-insensitive).

- `(apples|cats|baseballs)` matches one of the specified objects (case-insensitive).

- `\.` matches a period (escaped with a backslash).

- `$` asserts the end of the string.

This pattern will match sentences that have the specified first word, second word, third word, and end with a period.

Example usage:

```python

import re

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

sentences = [

'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 sentence in sentences:

if pattern.match(sentence):

print(f"Matched: {sentence}")

else:

print(f"Not matched: {sentence}")

```

Output:

```

Matched: Alice eats apples.

Matched: Bob pets cats.

Matched: Carol throws baseballs.

Matched: Alice throws Apples.

Matched: BOB EATS CATS.

Not matched: RoboCop eats apples.

Not matched: ALICE THROWS FOOTBALLS.

Not matched: Carol eats 7 cats.```

In the example, the sentences that match the pattern are printed as "Matched," while the ones that don't match are printed as "Not matched."