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. The re module provides functions and methods for working with regular expressions.

To utilize regex functionality in Python, you need to import the **re** module. It allows you to create regex objects that can be used for pattern matching, search, substitution, and other regex-related operations.

Here's an example demonstrating the usage of the **re** module to create a regex object:

pythonCopy code

import re pattern = r'\d+' # Regex pattern to match one or more digits regex\_obj = re.compile(pattern) # Now, you can use the regex object for various regex operations result = regex\_obj.search('There are 123 apples.') if result: print('Match found:', result.group()) else: print('No match found.')

In this example, the **re.compile()** function is used to create a regex object that matches one or more digits (**\d+**). The regex object (**regex\_obj**) can then be used to perform operations like searching for matches within a given string.

The **re** module provides various functions and methods, such as **search()**, **match()**, **findall()**, **sub()**, etc., that can be used with regex objects to accomplish different tasks related to pattern matching and manipulation.

Remember to import the **re** module at the beginning of your Python script to access the regex functionality it provides.

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

Ans : Raw strings (strings prefixed with r) are commonly used in regular expression (regex) objects in Python to simplify the handling of regex patterns. Raw strings treat backslashes (\) as literal characters, which is especially useful in regex patterns that often contain a significant number of backslashes.

In regex patterns, backslashes are frequently used to escape special characters or to represent special sequences. However, backslashes also have special meaning in regular Python strings as escape characters. This can lead to double escaping or confusion when writing regex patterns.

By using raw strings in regex objects, you can avoid the need for double escaping. Raw strings treat backslashes as literal characters, so you can write regex patterns more directly and clearly.

Here's an example to illustrate the use of raw strings in regex objects:

pythonCopy code

import re pattern = r'\d+' # Raw string representation of the regex pattern regex\_obj = re.compile(pattern) # Now, you can use the regex object for matching result = regex\_obj.search('There are 123 apples.') if result: print('Match found:', result.group()) else: print('No match found.')

In this example, the regex pattern **r'\d+'** is defined as a raw string. The raw string ensures that the backslash **\** is treated as a literal character and not an escape character. It allows the regex engine to interpret **\d** as a regex escape sequence representing a digit character.

Using raw strings in regex objects simplifies the expression of regex patterns and reduces the likelihood of errors caused by incorrect or unintended escaping.

While raw strings are not mandatory for regex objects, they are widely used for convenience and readability when working with complex regex patterns.

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

Ans : The search() method in Python's re module returns a match object if a match is found, or it returns None if no match is found.

The match object represents the match found by the **search()** method and provides various methods and attributes to access information about the match.

Here's an example that demonstrates the usage of the **search()** method and the possible return values:

pythonCopy code

import re pattern = r'apple' text = 'I have an apple.' match = re.search(pattern, text) if match: print('Match found:', match.group()) else: print('No match found.')

In this example, **re.search(pattern, text)** searches for the regex pattern **'apple'** within the text **'I have an apple.'**. If a match is found, the match object is returned and assigned to the variable **match**.

To determine if a match was found, we can check if **match** is truthy (i.e., not **None**). If it's truthy, we can access the matched string using **match.group()**.

If a match is found, the output will be:

sqlCopy code

Match found: apple

If no match is found, the output will be:

luaCopy code

No match found.

The return value of **search()** allows you to conditionally handle matches or non-matches and extract information from the match object when applicable.

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

Ans :To retrieve the actual strings that match a pattern from a match object in Python, you can use the group() method. The group() method returns the substring that was matched by the pattern.

Here's an example that demonstrates the usage of the **group()** method to retrieve the matched strings:

pythonCopy code

import re pattern = r'\d+' # Match one or more digits text = 'I have 123 apples and 456 oranges.' match = re.search(pattern, text) if match: matched\_string = match.group() print('Match found:', matched\_string) else: print('No match found.')

In this example, **re.search(pattern, text)** searches for the regex pattern **\d+** (one or more digits) within the text **'I have 123 apples and 456 oranges.'**. If a match is found, the match object is assigned to the variable **match**.

To retrieve the matched string, we can call **match.group()**. This will return the substring that matched the pattern.

If a match is found, the output will be:

Match found: 123

If no match is found, the output will be:

No match found.

Using the **group()** method allows you to access the actual strings that match the pattern and perform further operations or extract information from them. Note that if your pattern contains capturing groups, you can specify group numbers as arguments to **group()** to retrieve the matched strings within those specific 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 r'(\d\d\d)-(\d\d\d-\d\d\d\d)', which consists of capturing groups, the group numbering is as follows:

* Group 0: The entire match, including the hyphen. It covers the entire pattern.
* Group 1: It covers the first set of three digits surrounded by parentheses, **(\d\d\d)**.
* Group 2: It covers the second set of three digits followed by a hyphen and four digits, **(\d\d\d-\d\d\d\d)**.

Here's an example that demonstrates the usage of groups in the regex:

import re pattern = r'(\d\d\d)-(\d\d\d-\d\d\d\d)' text = 'Phone numbers: 123-456-7890 and 987-654-3210' match = re.search(pattern, text) if match: print('Group 0 (entire match):', match.group(0)) print('Group 1:', match.group(1)) print('Group 2:', match.group(2)) else: print('No match found.')

In this example, the regex **(\d\d\d)-(\d\d\d-\d\d\d\d)** is used to search for phone numbers in the given text. The **re.search()** function attempts to find a match.

If a match is found, we can access the matched strings within specific groups using the **group()** method with the respective group numbers as arguments.

The output will be:

Group 0 (entire match): 123-456-7890 Group 1: 123 Group 2: 456-7890

As you can see, **match.group(0)** returns the entire matched string, while **match.group(1)** returns the matched string within the first group, and **match.group(2)** returns the matched string within the second group.

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 ) and periods . have special meanings and are used as metacharacters to define grouping and matching behavior. If you want to match literal parentheses or periods in a regex pattern, you can escape them using a backslash \. This tells the regex engine to treat them as literal characters rather than metacharacters.

To include literal parentheses and periods in a regex pattern, you can use the backslash **\** to escape them. Here are the escape sequences for parentheses and periods:

* **\(**: Matches a literal opening parenthesis **(**.
* **\)**: Matches a literal closing parenthesis **)**.
* **\.**: Matches a literal period **.**.

Here's an example that demonstrates using escape sequences to match literal parentheses and periods:

pythonCopy code

import re pattern = r'\(Hello\)\.world\.' text = '(Hello).world.' match = re.search(pattern, text) if match: print('Match found:', match.group()) else: print('No match found.')

In this example, the regex pattern **r'\(Hello\)\.world\.' is used to search for the string** '(Hello).world.'**. The parentheses and period are escaped using backslashes to match them literally. The** re.search()` function attempts to find a match.

If a match is found, the output will be:

sqlCopy code

Match found: (Hello).world.

By escaping parentheses and periods with backslashes, you can specify that they should be treated as literal characters in the regex pattern.

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 re module returns either a list of strings or a list of string tuples depending on the presence of capturing groups in the regular expression pattern.

Here's how the return value of **findall()** is determined:

1. If the regex pattern contains no capturing groups (i.e., no parentheses), **findall()** returns a list of strings. Each element in the list represents a complete match of the pattern.
2. If the regex pattern contains capturing groups (i.e., parentheses are used to define groups), **findall()** returns a list of string tuples. Each tuple in the list corresponds to a complete match of the pattern, and each element within the tuple represents the substring matched by a capturing group.

Let's see an example to illustrate this behavior:

import re pattern = r'(\d+)-(\w+)' text = '123-abc 456-def 789-ghi' matches = re.findall(pattern, text) print(matches)

In this example, the regex pattern **(\d+)-(\w+)** is used to search for matches in the given text. The pattern consists of two capturing groups: **(\d+)** and **(\w+)**. The **re.findall()** function is used to find all matches.

If the regex pattern contains no capturing groups, the output will be:

['123', '456', '789']

Since there are no capturing groups in this case, **findall()** returns a list of strings representing the complete matches of the pattern (in this case, the numbers).

If the regex pattern contains capturing groups, the output will be:

[('123', 'abc'), ('456', 'def'), ('789', 'ghi')]

Since the pattern contains capturing groups, **findall()** returns a list of string tuples. Each tuple contains the matched substring of the complete match and the substring matched by each capturing group.

By using capturing groups in the regex pattern, you can extract specific portions of the matches as separate elements within the resulting tuples.

So, the return value of **findall()** depends on whether the regex pattern includes capturing groups or not.

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 alternatives or choices within a regex pattern. The | operator allows you to match either the expression before it or the expression after it.

Here's how the **|** character works in regular expressions:

* **expr1 | expr2**: Matches either **expr1** or **expr2**.

The pipe operator can be used to create multiple alternative patterns within a single regex. It behaves like a logical OR operation, matching either the left-hand side or the right-hand side of the **|** operator.

For example, consider the following regex pattern: **cat|dog**. It matches either the string "cat" or the string "dog". If the input text contains either "cat" or "dog", the pattern will find a match.

Here's an example using Python's **re** module to demonstrate the usage of the **|** operator:

import re pattern = r'cat|dog' text = 'I have a cat and a dog.' matches = re.findall(pattern, text) print(matches)

In this example, the regex pattern **cat|dog** is used to search for occurrences of either "cat" or "dog" in the given text. The **re.findall()** function returns a list of all matches found.

The output will be:

['cat', 'dog']

Since the text contains both "cat" and "dog", both alternatives are matched and returned in the resulting list.

The **|** character in regular expressions provides a way to define choices or alternatives within a pattern, allowing you to match different possibilities based on your specific matching requirements.

9. In regular expressIf you are referring to the character "word" in regular expressions, it typically represents a word boundary. In regular expression syntax, the "word" character, represented by \b, is a metacharacter that matches the position between a word character and a non-word character.

Here are some key aspects of the **\b** word boundary in regular expressions:

* **\b** Matches the position at the beginning or end of a word.
* It does not match any characters directly, only positions.
* It allows you to search for whole words, as opposed to substrings within larger words.
* It is useful when you want to match a specific word as a standalone entity.

For example, consider the regular expression pattern **\bcat\b**. This pattern matches the word "cat" only when it appears as a separate word, not as part of a larger word like "concatenate" or "category".

ions, what does the character stand for?

Ans :

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

Ans : In regular expressions, the + and \* characters are known as quantifiers and are used to specify the repetition of the preceding element in the regex pattern. However, they have different meanings and behaviors:

* **+** (Plus): The **+** quantifier matches one or more occurrences of the preceding element.

For example, **a+** matches one or more consecutive occurrences of the letter "a". It will match "a", "aa", "aaa", and so on, but it will not match an empty string or any string without at least one "a".

* **\*** (Asterisk): The **\*** quantifier matches zero or more occurrences of the preceding element.

For example, **a\*** matches zero or more consecutive occurrences of the letter "a". It will match an empty string, "a", "aa", "aaa", and any string containing "a" zero or more times.

To illustrate the difference, consider the regex pattern **ca+t**. This pattern will match "cat", "caat", "caaat", and so on, but it will not match "ct" because there must be at least one occurrence of "a" between "c" and "t".

On the other hand, if we have the regex pattern **ca\*t**, it will match "ct" (zero occurrences of "a"), "cat", "caat", "caaat", and any string that contains "c" followed by "t" with or without "a" in between.

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

Ans : In regular expressions, the expressions {4} and {4,5} are used as quantifiers to specify the exact number or a range of repetitions for the preceding element in the pattern. Here's the difference between them:

* **{4}**: This quantifier matches exactly four occurrences of the preceding element. It specifies an exact repetition count.

For example, **a{4}** matches exactly four consecutive occurrences of the letter "a". It will match "aaaa" but not "aaa" or "aaaaa".

* **{4,5}**: This quantifier matches a range of repetitions for the preceding element, between four and five occurrences (inclusive). It specifies a minimum and maximum repetition count.

For example, **a{4,5}** matches between four and five consecutive occurrences of the letter "a". It will match "aaaa" and "aaaaa", but not "aaa" or "aaaaaa".

In summary, **{4}** matches exactly four repetitions, while **{4,5}** matches between four and five repetitions (inclusive).

Here's an example using Python's **re** module to demonstrate the usage of **{4}** and **{4,5}** quantifiers:

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

Ans :   
In regular expressions, shorthand character classes such as \d, \w, and \s are predefined character classes that represent commonly used sets of characters. They provide a shorthand way to match specific categories of characters within a regex pattern.

Here's what each shorthand character class signifies:

* **\d**: Matches any digit character. It is equivalent to the character class **[0-9]**.

For example, **\d** will match any single digit character from 0 to 9.

* **\w**: Matches any word character. It includes alphanumeric characters (letters and digits) and underscores (**\_**). It is equivalent to the character class **[a-zA-Z0-9\_]**.

For example, **\w** will match any letter (uppercase or lowercase), digit, or underscore character.

* **\s**: Matches any whitespace character. It includes spaces, tabs, newlines, and other whitespace characters.

For example, **\s** will match spaces, tabs, newlines, and other whitespace characters.

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

Ans : In regular expressions, the shorthand character classes \D, \W, and \S are used to match characters that are NOT within certain categories. They represent the negations of the \d, \w, and \s shorthand character classes, respectively.

Here's what each shorthand character class signifies:

* **\D**: Matches any character that is not a digit. It is equivalent to the negation of the character class **[0-9]**.

For example, **\D** will match any character that is not a digit (0-9).

* **\W**: Matches any character that is not a word character. It is equivalent to the negation of the character class **[a-zA-Z0-9\_]**.

For example, **\W** will match any character that is not a letter, digit, or underscore.

* **\S**: Matches any character that is not a whitespace character.

For example, **\S** will match any character that is not a space, tab, newline, or other whitespace character.

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

Ans :   
The expressions .\*? and .\*? are both examples of non-greedy or lazy quantifiers in regular expressions. However, they represent slightly different matching behaviors:

* **.\*?**: This non-greedy quantifier matches as few characters as possible to fulfill the overall pattern.

For example, **a.\*?b** matches the shortest possible substring that starts with "a" and ends with "b". It will match "ab", "aab", "aaab", etc., as long as "a" and "b" are present.

* **.\*?**: This non-greedy quantifier also matches as few characters as possible, but it has a different default behavior when used on its own.

For example, **.\*?** by itself will match an empty string or the shortest possible substring of any characters.

The difference between the two expressions is primarily in how they interact with surrounding patterns. The specific behavior depends on the context in which they are used.

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 a regular expression, you can use the range notation within the character class.

Here's the syntax to match numbers (0-9) and lowercase letters (a-z) using a character class:

csharpCopy code

[0-9a-z]

Within the square brackets **[ ]**, the range **0-9** specifies all digits (0 to 9), and the range **a-z** specifies all lowercase letters (a to z).

You can include this character class within your regular expression pattern to match any occurrence of a number or lowercase letter.

Here's an example using Python's **re** module:

pythonCopy code

import re pattern = r'[0-9a-z]' text = 'abc123DEF456' matches = re.findall(pattern, text) print(matches)

The output will be:

cssCopy code

['a', 'b', 'c', '1', '2', '3', '4', '5', '6']

In this example, the pattern **[0-9a-z]** is used to search for matches in the given text. The **re.findall()** function returns a list of all matches found.

As you can see, the character class **[0-9a-z]** matches each occurrence of a number or lowercase letter in the text.

By using a character class with the appropriate range, you can specify a pattern that matches both numbers and lowercase letters in a concise and effective way.

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 flag when compiling the regex pattern. These flags allow the pattern to match characters regardless of their case.

Here's the procedure for making a regular expression case-insensitive:

1. Import the **re** module: Start by importing the **re** module, which provides the functions and flags for working with regular expressions in Python.
2. Compile the regex pattern with the case-insensitive flag: Use the **re.compile()** function to compile your regex pattern. Pass the **re.IGNORECASE** flag or the **re.I** flag as the second argument to the **re.compile()** function.
3. Use the compiled regex object for matching: With the compiled regex object, you can use its various methods (**search()**, **match()**, **findall()**, etc.) to perform case-insensitive matching.

Here's an example that demonstrates making a regular expression case-insensitive using the **re.IGNORECASE** flag:

import re pattern = r'hello' text = 'Hello World' regex\_obj = re.compile(pattern, re.IGNORECASE) match = regex\_obj.search(text) if match: print('Match found:', match.group()) else: print('No match found.')

The output will be:

Match found: Hello

In this example, the regex pattern **'hello'** is compiled using the **re.compile()** function with the **re.IGNORECASE** flag. The **re.IGNORECASE** flag makes the pattern case-insensitive.

When the **search()** method is called on the regex object with the text **'Hello World'**, it finds a match even though the case of the text doesn't exactly match the pattern.

You can also use the shorthand flag **re.I** instead of **re.IGNORECASE**:

regex\_obj = re.compile(pattern, re.I)

Both **re.IGNORECASE** and **re.I** flags have the same effect of making the regular expression case-insensitive.

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 (\n). It is often referred to as a wildcard character that represents any single character.

However, when the **re.DOTALL** flag is passed as the second argument to the **re.compile()** function, the behavior of the dot character changes. With **re.DOTALL**, the dot character matches any character, including a newline (**\n**).

Here's an example to illustrate the default behavior of the dot character and the effect of the **re.DOTALL** flag:

import re pattern = r'.' text = 'Hello\nWorld' regex\_obj = re.compile(pattern) match = regex\_obj.search(text) print('Default behavior:', match.group()) regex\_obj\_dotall = re.compile(pattern, re.DOTALL) match\_dotall = regex\_obj\_dotall.search(text) print('With re.DOTALL:', match\_dotall.group())

The output will be:

Default behavior: H With re.DOTALL: H e

In this example, the dot character **.** is used as the regex pattern. The **re.compile()** function is called both with and without the **re.DOTALL** flag.

By default, without the **re.DOTALL** flag, the dot character matches only the first character **'H'**, as it doesn't include newlines.

However, when the **re.DOTALL** flag is used, the dot character matches all characters, including the newline. In this case, it matches the letter **'H'** and the newline character **'\n'**.

The **re.DOTALL** flag is useful when you want the dot character to match any character, including newlines. It enables you to match across multiple lines in a text.

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 you execute numRegex.sub('X', '11 drummers, 10 pipers, five rings, 4 hen'), the sub() method will substitute all occurrences of the regex pattern \d+ (one or more digits) in the input string with the replacement string 'X'.

Here's the result:

pythonCopy code

import re numRegex = re.compile(r'\d+') result = numRegex.sub('X', '11 drummers, 10 pipers, five rings, 4 hen') print(result)

Output:

Copy code

X drummers, X pipers, five rings, X hen

In this example, the **sub()** method replaces all the numeric substrings with **'X'**. So, **'11'** and **'10'** are replaced with **'X'**, while **'5'** and **'4'** are unaffected because they do not match the pattern **\d+**.

Therefore, the resulting string is **'X drummers, X pipers, five rings, X hen'**, where all the numeric substrings are replaced with **'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() allows you to use the verbose mode in regular expressions. In verbose mode, you can write more readable and well-structured regular expressions by adding comments and whitespace without affecting the pattern matching.

Here's what **re.VERBOSE** enables you to do:

1. Add comments: You can include comments within the regex pattern to explain the purpose of certain parts or provide documentation. Comments start with the **#** character and continue until the end of the line.
2. Ignore whitespace: Whitespace characters (spaces, tabs, and newlines) within the regex pattern are ignored, allowing you to format and structure the pattern more clearly.
3. Enable multiline patterns: You can split the pattern into multiple lines for better readability.

Using **re.VERBOSE** can make complex regex patterns more understandable and maintainable. It allows you to add comments, organize the pattern with whitespace, and break it into multiple lines without affecting the matching behavior.

Here's an example to demonstrate the usage of **re.VERBOSE**:

import re pattern = r""" \d{3} # Match three digits - # Match a hyphen \d{4} # Match four digits """ text = 'Phone number: 123-4567' regex\_obj = re.compile(pattern, re.VERBOSE) match = regex\_obj.search(text) if match: print('Match found:', match.group()) else: print('No match found.')

Output:

Match found: 123-4567

In this example, the pattern to match a phone number is written in verbose mode by passing **re.VERBOSE** as the second argument to **re.compile()**. The pattern is split into multiple lines and includes comments to explain each part of the pattern.

Using **re.VERBOSE** does not change the pattern itself; it only affects the interpretation of the pattern during compilation. It allows you to write more readable and organized regular expressions by including comments and whitespace.

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, you can use the following regular expression:

pythonCopy code

import re pattern = r'^\d{1,3}(,\d{3})\*$' numbers = ['42', '1,234', '6,368,745', '12,34,567', '1234'] for number in numbers: match = re.match(pattern, number) if match: print('Match found:', match.group()) else: print('No match found.')

Output:

sqlCopy code

Match found: 42 Match found: 1,234 Match found: 6,368,745 No match found. No match found.

In this example, the pattern **^\d{1,3}(,\d{3})\*$** is used to match a number with commas for every three digits.

Explanation of the pattern:

* **^** and **$** match the start and end of the string, ensuring that the entire string is matched.
* **\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.
* The pattern ensures that the entire string consists of groups of one to three digits separated by commas.

The **re.match()** function is used to check if the pattern matches the given numbers. If a match is found, it is printed. Otherwise, a "No match found" message is printed.

As you can see, the pattern matches numbers '42', '1,234', and '6,368,745', which have commas separating every three digits. It does not match '12,34,567' (which has only two digits between the commas) or '1234' (which lacks commas).

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 will always be one word beginning with a capital letter, you can use the following regular expression:

import re pattern = 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: match = re.match(pattern, name) if match: print('Match found:', match.group()) else: print('No match found.')

Output:

Match found: Haruto Watanabe Match found: Alice Watanabe Match found: RoboCop Watanabe No match found. No match found. No match found. No match found.

In this example, the pattern **^[A-Z][a-zA-Z]\* Watanabe$** is used to match the full name of someone with the last name Watanabe.

Explanation of the pattern:

* **^** and **$** match the start and end of the string, ensuring that the entire string is matched.
* **[A-Z]** matches a single 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).
* **Watanabe** matches the last name specifically as "Watanabe".

The pattern ensures that the entire string consists of a first name followed by a space and the last name "Watanabe". The first name must start with a capital letter, and the last name must be exactly "Watanabe".

The **re.match()** function is used to check if the pattern matches the given names. If a match is found, it is printed. Otherwise, a "No match found" message is printed.

As you can see, the pattern matches names 'Haruto Watanabe', 'Alice Watanabe', and 'RoboCop Watanabe', which follow the specified format. It does not match names that do not meet the requirements, such as 'haruto Watanabe', 'Mr. Watanabe', 'Watanabe', or 'Haruto watanabe'.

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 write a case-insensitive regex that matches a sentence with specific word combinations, you can use the following pattern:

import re pattern = r'^(Alice|Bob|Carol) (eats|pets|throws) (apples|cats|baseballs)\.$' sentences = [ 'Alice eats apples.', 'Bob pets cats.', 'Carol throws baseballs.', 'alice eats apples.', 'Bob throws cats.', 'Dave pets apples.', 'Carol eats oranges.', ] for sentence in sentences: match = re.match(pattern, sentence, re.IGNORECASE) if match: print('Match found:', match.group()) else: print('No match found.')

Output:

Match found: Alice eats apples. Match found: Bob pets cats. Match found: Carol throws baseballs. Match found: alice eats apples. Match found: Bob throws cats. No match found. No match found.

In this example, the pattern **^(Alice|Bob|Carol) (eats|pets|throws) (apples|cats|baseballs)\.$** is used to match the specified sentence structure.

Explanation of the pattern:

* **^** and **$** match the start and end of the string, ensuring that the entire string is matched.
* **(Alice|Bob|Carol)** matches either "Alice", "Bob", or "Carol" as the first word.
* **(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.
* **\.**, escaped with a backslash, matches the period at the end of the sentence.

The pattern allows for case-insensitive matching using the **re.IGNORECASE** flag.

The **re.match()** function is used to check if the pattern matches the given sentences. If a match is found, it is printed. Otherwise, a "No match found" message is printed.

As you can see, the pattern matches sentences that follow the specified structure, regardless of the case. It does not match sentences that do not meet the requirements.