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 provided by the `re` module. The `re` module stands for "regular expression," and it provides functions and methods to work with regular expressions in Python.

To use regular expressions in Python, you need to import the `re` module, and then you can create regular expression objects using various functions like `re.compile()` or directly use the functions like `re.match()`, `re.search()`, `re.findall()`, and others.

For example:

import re

# Creating a regular expression object using re.compile()

pattern = re.compile(r'\d{3}-\d{2}-\d{4}')

# Using re.search() to find a match

text = 'My social security number is 123-45-6789.'

match = pattern.search(text)

if match:

print('Found SSN:', match.group())

In this example, the regular expression pattern `\d{3}-\d{2}-\d{4}` is created using `re.compile()`, and then `re.search()` is used to find a match in the given text. The `re` module is responsible for handling regular expressions and provides powerful tools for pattern matching and text manipulation.

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

Ans: Raw strings (strings prefixed with `r`) are often used with regular expressions in Python to avoid potential issues with backslashes. In regular expressions, backslashes `\` are commonly used as escape characters to represent special characters or to create character classes. However, Python strings also use backslashes as escape characters, which can lead to conflicts and unexpected behavior when working with regular expressions.

Using raw strings with regular expressions helps prevent accidental misinterpretation of backslashes, making it easier to write and read regular expressions. In a raw string, backslashes are treated literally and not as escape characters. This means that when using raw strings with regular expressions, you don't need to escape backslashes twice.

For example, consider a regular expression that matches a valid email address:

import re

pattern = re.compile(r'\b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b')

In this example, the `r` before the string indicates that it's a raw string. Without the `r`, you would have to escape the backslashes twice:

pattern = re.compile('\\b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\\.[A-Z|a-z]{2,}\\b')

Using raw strings makes regular expressions more readable and less error-prone, as you can write the patterns more naturally without worrying about double-escaping backslashes.Remember that using raw strings is not mandatory, but it is a recommended best practice when working with regular expressions in Python to avoid potential issues related to backslashes and escaping.

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

Ans: The `search()` method of the `re` module in Python returns a match object if a match is found, or `None` if no match is found.

Syntax:

re.search(pattern, string, flags=0)

Parameters:

- `pattern`: The regular expression pattern to search for.

- `string`: The input string where the search will be performed.

- `flags` (optional): Special flags that control the behavior of the search.

Return value:

- If a match is found, `search()` returns a match object, which contains information about the match and allows you to extract matched substrings and other details.

- If no match is found, `search()` returns `None`.

Example:

import re

pattern = r'\d{3}-\d{2}-\d{4}'

text = 'My SSN is 123-45-6789.'

match = re.search(pattern, text)

if match:

print('Found SSN:', match.group())

else:

print('No SSN found.')

In this example, the `search()` method is used to search for a social security number (SSN) pattern in the text. If the pattern is found in the `text`, a match object is returned, and we print the matched SSN using `match.group()`. If there's no match, the method returns `None`, and the program prints "No SSN found."It's important to check the return value of `search()` before using the match object to avoid potential errors when no match is found in the string.

1. 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 or the `groups()` method. The `group()` method returns the entire matched substring, while the `groups()` method returns a tuple containing the matched substrings for each capturing group in the pattern.

Here's how you can use these methods:

a). `group()`: Returns the entire matched substring.

import re

pattern = r'\d{3}-\d{2}-\d{4}'

text = 'My SSN is 123-45-6789.'

match = re.search(pattern, text)

if match:

matched\_string = match.group()

print('Found SSN:', matched\_string)

else:

print('No SSN found.')

b). `groups()`: Returns a tuple containing the matched substrings for each capturing group.

import re

pattern = r'(\d{3})-(\d{2})-(\d{4})'

text = 'My SSN is 123-45-6789.'

match = re.search(pattern, text)

if match:

matched\_groups = match.groups()

print('Found SSN groups:', matched\_groups)

else:

print('No SSN found.')

In this example, the first pattern uses `\d{3}-\d{2}-\d{4}` to match a social security number (SSN) pattern, and the second pattern `(\d{3})-(\d{2})-(\d{4})` uses capturing groups to extract each part of the SSN separately. When using `group()`, it returns the entire matched SSN (e.g., `'123-45-6789'`). When using `groups()`, it returns a tuple with the matched substrings for each capturing group (e.g., `('123', '45', '6789')`).

1. 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)'`, the parentheses `()` define capturing groups. The numbers inside the parentheses represent the order of the capturing groups, starting from 1. The `group(0)` is a special case and represents the entire matched substring.

Here's a breakdown of the groups:

- `group(0)`: Represents the entire matched substring (i.e., the full social security number).

- `group(1)`: Represents the first capturing group `(\d\d\d)`, which matches three digits in sequence before the hyphen (e.g., the first three digits of the social security number).

- `group(2)`: Represents the second capturing group `(\d\d\d-\d\d\d\d)`, which matches three digits, followed by a hyphen, and then four digits (e.g., the last seven digits of the social security number).

Here's an example of how to use these groups:

import re

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

text = 'My SSN is 123-456-7890.'

match = re.search(pattern, text)

if match:

# Full SSN

full\_ssn = match.group(0)

print('Full SSN:', full\_ssn)

# First capturing group (first three digits)

first\_group = match.group(1)

print('First three digits:', first\_group)

# Second capturing group (last seven digits)

second\_group = match.group(2)

print('Last seven digits:', second\_group)

else:

print('No SSN found.')

Output:

Full SSN: 123-456-7890

First three digits: 123

Last seven digits: 456-7890

In this example, `group(0)` represents the full SSN `'123-456-7890'`, `group(1)` represents the first three digits `'123'`, and `group(2)` represents the last seven digits `'456-7890'`.

1. 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 (dots) have special meanings, but sometimes you might need to match them as literal characters in the input string. To tell the regular expression to interpret parentheses and periods as literal characters, you can use backslashes `\` to escape them.

Here's how you can use backslashes to match real parentheses and periods:

A) For matching real parentheses, use `\(` to match an opening parenthesis and `\)` to match a closing parenthesis.

Example:

import re

text = "I have (some parentheses) in this (text)."

pattern = r'\(.\*?\)'

matches = re.findall(pattern, text)

print(matches)

# Output: ['(some parentheses)', '(text)']

In this example, the regular expression `r'\(.\*?\)'` matches any substring enclosed in parentheses.

B). For matching real periods, use `\.` to match a literal dot.

Example:

import re

text = "The price is $3.99. Buy now."

pattern = r'\$[0-9]+\.'

matches = re.findall(pattern, text)

print(matches)

# Output: ['$3.99.']

In this example, the regular expression `r'\$[0-9]+\.'` matches a substring that starts with a dollar sign `$`, followed by one or more digits `[0-9]+`, and ends with a literal dot `.`. By using backslashes `\`, you can escape special characters in regular expressions, including parentheses and periods, and match them as literal characters in the input string.

1. 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 based on the presence of capturing groups in the regular expression pattern.

When the regular expression pattern contains at least one capturing group (defined using parentheses `()`), the `findall()` method returns a list of string tuples. Each tuple in the list represents a match, and each element of the tuple corresponds to the substring matched by a capturing group in the pattern.

On the other hand, if the regular expression pattern has no capturing groups, the `findall()` method returns a simple list of strings. Each element in the list represents the entire matched substring.

Here's an example to illustrate the difference:

import re

# Pattern with a capturing group

pattern\_with\_group = r'(\d{3})-(\d{2})-(\d{4})'

text = 'My SSN is 123-45-6789, and yours is 987-65-4321.'

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

print('Matches with capturing group:', matches\_with\_group)

# Output: [('123', '45', '6789'), ('987', '65', '4321')]

# Pattern without a capturing group

pattern\_without\_group = r'\d{3}-\d{2}-\d{4}'

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

print('Matches without capturing group:', matches\_without\_group)

# Output: ['123-45-6789', '987-65-4321']

In the example above, `pattern\_with\_group` contains capturing groups, so `findall()` returns a list of string tuples, where each tuple represents the matched substrings for each capturing group in the pattern.On the other hand, `pattern\_without\_group` has no capturing groups, so `findall()` returns a simple list of strings representing the entire matched substrings.

If you want `findall()` to return a list of entire matched substrings even with capturing groups, you can use a non-capturing group `(?:...)` or convert the capturing group to a non-capturing group by adding `?:` at the beginning of the group. For example:

pattern\_with\_non\_capturing\_group = r'(?:\d{3})-(?:\d{2})-(?:\d{4})'

With this modified pattern, `findall()` will return a list of entire matched substrings instead of string tuples.

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

Ans: In regular expressions, the `|` (vertical bar or pipe) character is used as the "alternation" or "OR" operator. It allows you to specify multiple alternative patterns, and the regular expression will match any of the provided alternatives.

The syntax for using the `|` operator is as follows:

pattern1 | pattern2 | pattern3 | ...

In this syntax, `pattern1`, `pattern2`, `pattern3`, and so on, represent different regular expression patterns. The `|` operator separates these patterns, indicating that any of them can be matched for a successful overall match.

Example:

Let's say we want to find occurrences of either "cat" or "dog" in a given text:

import re

text = "I have a cat and a dog."

pattern = r'cat|dog'

matches = re.findall(pattern, text)

print(matches)

# Output: ['cat', 'dog']

In this example, the regular expression pattern `r'cat|dog'` contains two alternatives separated by `|`. The `findall()` method will search for either "cat" or "dog" in the `text` and return a list of all occurrences found.In summary, the `|` character in regular expressions is used for alternation, allowing you to specify multiple alternative patterns to match in the input text. If any of the alternatives match, the overall match is considered successful.

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

Ans: In regular expressions, the `.` (dot) character is called the "dot" or "period" metacharacter, and it has a special meaning. It represents any single character except a newline character (`\n`). In other words, the dot `.` matches any character in the input string except for the newline.

For example:

import re

text = "cat hat bat"

pattern = r'.at'

matches = re.findall(pattern, text)

print(matches)

# Output: ['cat', 'hat', 'bat']

In this example, the regular expression pattern `r'.at'` contains the dot metacharacter, which matches any single character followed by "at". When applied with `findall()` to the `text`, it finds all occurrences of three-letter words ending with "at" and returns a list of matches: `['cat', 'hat', 'bat']`.Overall, the dot `.` is a powerful and commonly used metacharacter in regular expressions, allowing you to match any character in the input string (except for a newline) and creating more flexible and dynamic patterns.

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

Ans: In regular expressions, the `+` and `\*` characters are quantifiers that specify how many times the preceding character or group should be matched. The main difference between them lies in the number of occurrences they allow:

i). `+` (Plus): This quantifier matches one or more occurrences of the preceding character or group. It requires the preceding character or group to appear at least once, but it can also match if it appears multiple times.

Example:

import re

text = "aaa ab aab"

pattern = r'a+b'

matches = re.findall(pattern, text)

print(matches)

# Output: ['aaa', 'ab', 'aab']

In this example, the regular expression pattern `r'a+b'` matches sequences of one or more consecutive "a" characters followed by a "b". It matches `'aaa'`, `'ab'`, and `'aab'`.

ii). `\*` (Asterisk): This quantifier matches zero or more occurrences of the preceding character or group. It allows the preceding character or group to appear any number of times, including zero.

Example:

import re

text = "cat ct caaat"

pattern = r'ca\*t'

matches = re.findall(pattern, text)

print(matches)

# Output: ['cat', 'ct', 'caaat']

In this example, the regular expression pattern `r'ca\*t'` matches sequences of "c" followed by zero or more "a" characters and then "t". It matches `'cat'`, `'ct'`, and `'caaat'`.

In summary:

- `+` matches one or more occurrences of the preceding character or group.

- `\*` matches zero or more occurrences of the preceding character or group.

Both quantifiers are powerful tools for creating flexible and dynamic regular expression patterns that can handle various repetitions of characters or groups in the input string.

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

Ans: In regular expressions, `{4}` and `{4,5}` are both quantifiers used to specify the number of occurrences of the preceding character or group. The main difference between them lies in the exact number of occurrences they allow:

a). `{4}`: This quantifier matches exactly four occurrences of the preceding character or group.

Example:

import re

text = "12345 1234 123456 123"

pattern = r'\d{4}'

matches = re.findall(pattern, text)

print(matches)

# Output: ['1234', '1234']

In this example, the regular expression pattern `\d{4}` matches exactly four consecutive digits in the `text`. It matches `'1234'` twice, which are the only occurrences of four consecutive digits.

b). `{4,5}`: This quantifier matches between four and five occurrences of the preceding character or group. It allows the preceding character or group to appear at least four times, but it can also match if it appears five times.

Example:

import re

text = "12345 1234 123456 123 1234567"

pattern = r'\d{4,5}'

matches = re.findall(pattern, text)

print(matches)

# Output: ['12345', '1234', '12345', '12345']

In this example, the regular expression pattern `\d{4,5}` matches sequences of four or five consecutive digits in the `text`. It matches `'12345'` (appearing twice), `'1234'`, and the first five digits of `'1234567'`.

In summary:

- `{4}` matches exactly four occurrences of the preceding character or group.

- `{4,5}` matches between four and five occurrences of the preceding character or group.

Both quantifiers are useful for defining specific repetition patterns in regular expressions, and they provide flexibility in matching various lengths of sequences in the input string.

1. 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 classes represented by special escape sequences. They allow you to match specific types of characters in the input string without explicitly listing all possible characters. The most common shorthand character classes are:

a.) `\d`: Represents any digit character. It is equivalent to the character class `[0-9]`.

Example:

import re

text = "The price is $3.99."

pattern = r'\d+'

matches = re.findall(pattern, text)

print(matches)

# Output: ['3', '99']

In this example, the regular expression pattern `\d+` matches one or more consecutive digit characters in the `text`. It matches `'3'` and `'99'`, which are the digit parts of the price.

b.) `\w`: Represents any word character. It matches alphanumeric characters (letters, digits, and underscores). It is equivalent to the character class `[a-zA-Z0-9\_]`.

Example:

import re

text = "Hello, world! This is an example."

pattern = r'\w+'

matches = re.findall(pattern, text)

print(matches)

# Output: ['Hello', 'world', 'This', 'is', 'an', 'example']

In this example, the regular expression pattern `\w+` matches one or more consecutive word characters in the `text`. It matches all the words in the text.

c.) `\s`: Represents any whitespace character. It matches spaces, tabs, newlines, and other whitespace characters.

Example:

import re

text = "The quick\tbrown\nfox jumps over the lazy dog."

pattern = r'\s+'

matches = re.findall(pattern, text)

print(matches)

# Output: [' ', '\t', '\n', ' ', ' ', ' ']

In this example, the regular expression pattern `\s+` matches one or more consecutive whitespace characters in the `text`. It matches spaces, tabs, and newlines.These shorthand character classes are useful for creating more concise and readable regular expressions when you need to match specific types of characters in the input string. They provide a convenient way to represent common character groups and simplify the process of pattern matching.

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

Ans: In regular expressions, the `\D`, `\W`, and `\S` are shorthand character classes that have special meanings. They represent negations of their counterparts `\d`, `\w`, and `\s`, respectively. These shorthand character classes allow you to match characters that are not of a specific type without explicitly listing all possible characters.

1. `\D`: Represents any non-digit character. It matches any character that is not a digit. It is equivalent to the character class `[^0-9]`.

Example:

import re

text = "The price is $3.99."

pattern = r'\D+'

matches = re.findall(pattern, text)

print(matches)

# Output: ['The price is $', '.']

In this example, the regular expression pattern `\D+` matches one or more consecutive non-digit characters in the `text`. It matches `'The price is $'` and `'.'`, which are all the non-digit parts of the text.

B.)`\W`: Represents any non-word character. It matches any character that is not a word character (non-alphanumeric characters and underscores). It is equivalent to the character class `[^a-zA-Z0-9\_]`.

Example:

import re

text = "Hello, world! This is an example."

pattern = r'\W+'

matches = re.findall(pattern, text)

print(matches)

# Output: [', ', '! ', ' ', ' ', '.']

In this example, the regular expression pattern `\W+` matches one or more consecutive non-word characters in the `text`. It matches all the punctuation and space between words.

C.) `\S`: Represents any non-whitespace character. It matches any character that is not a whitespace character. It is equivalent to the character class `[^ \t\n\r\f\v]`.

Example:

import re

text = "The quick\tbrown\nfox jumps over the lazy dog."

pattern = r'\S+'

matches = re.findall(pattern, text)

print(matches)

# Output: ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog.']

In this example, the regular expression pattern `\S+` matches one or more consecutive non-whitespace characters in the `text`. It matches all the words and punctuation, excluding the whitespace. These shorthand character classes with negations (`\D`, `\W`, and `\S`) are useful when you want to match characters that are not of a specific type, and they provide a concise way to represent these character groups in regular expressions.

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

Ans: In regular expressions, `.\*?` and `.\*` are both quantifiers used to match zero or more occurrences of any character in a non-greedy or greedy manner, respectively. The main difference between them lies in their behavior when it comes to matching characters in the input string.

i). `.\*?` (Non-Greedy or Lazy Quantifier):

The `.\*?` quantifier is a non-greedy or lazy quantifier. It matches as few characters as possible while still allowing the rest of the regular expression to match. It will stop as soon as the subsequent part of the regular expression can match, even if it means matching fewer characters.

Example:

import re

text = "Hello, world!"

pattern = r'H.\*?o'

match = re.search(pattern, text)

print(match.group())

# Output: 'Hello'

In this example, the regular expression pattern `r'H.\*?o'` matches the substring starting with 'H' and ending with the first occurrence of 'o' that follows 'H'. It uses a non-greedy quantifier `.\*?`, so it matches as few characters as possible between 'H' and 'o', resulting in the match `'Hello'`.

ii). `.\*` (Greedy Quantifier):

The `.\*` quantifier is a greedy quantifier. It matches as many characters as possible while still allowing the rest of the regular expression to match. It will try to match as many characters as it can before backtracking to satisfy the subsequent part of the regular expression.

Example:

import re

text = "Hello, world!"

pattern = r'H.\*o'

match = re.search(pattern, text)

print(match.group())

# Output: 'Hello, world'

In this example, the regular expression pattern `r'H.\*o'` matches the substring starting with 'H' and ending with the last occurrence of 'o' in the `text`. It uses a greedy quantifier `.\*`, so it matches as many characters as possible between 'H' and the last 'o', resulting in the match `'Hello, world'`.

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 predefined character classes `\d` and `[a-z]` together within a single character class. The combination of `\d` and `[a-z]` will match any digit from 0 to 9 and any lowercase letter from 'a' to 'z', respectively.

Here's the syntax:

import re

text = "abc123xyz456"

pattern = r'[a-z\d]+'

matches = re.findall(pattern, text)

print(matches)

# Output: ['abc123xyz456']

In this example, the regular expression pattern `r'[a-z\d]+'` matches one or more occurrences of either lowercase letters or digits in the `text`. It matches the entire alphanumeric sequence `'abc123xyz456'`.

In summary, you can use `[a-z\d]` within a character class to match both lowercase letters and digits in a regular expression. The `[a-z]` part matches any lowercase letter, and the `\d` part matches any digit from 0 to 9. The `+` quantifier ensures that one or more such characters are matched in the input string.

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` (or `re.I`) flag when compiling the regular expression pattern. This flag allows the pattern to match characters regardless of their case.

Here's the procedure:

a) Import the `re` module.

b) Define your regular expression pattern, and use the `re.IGNORECASE` flag as the second argument when calling the `re.compile()` function.

Example:

import re

text = "Hello, hello, hELLo, HeLLo!"

# Regular expression pattern with IGNORECASE flag

pattern = re.compile(r'hello', re.IGNORECASE)

matches = pattern.findall(text)

print(matches)

# Output: ['Hello', 'hello', 'hELLo', 'HeLLo']

In this example, the regular expression pattern `r'hello'` will match the word "hello" in the input string, regardless of its case. The `re.IGNORECASE` flag allows the pattern to match variations of "hello" in different cases, resulting in the output `['Hello', 'hello', 'hELLo', 'HeLLo']`.

You can also use the `(?i)` inline flag within the pattern to achieve case insensitivity:

import re

text = "Hello, hello, hELLo, HeLLo!"

# Regular expression pattern with inline flag for case insensitivity

pattern = re.compile(r'(?i)hello')

matches = pattern.findall(text)

print(matches)

# Output: ['Hello', 'hello', 'hELLo', 'HeLLo']

Both methods (using `re.IGNORECASE` flag or `(?i)` inline flag) have the same effect of making the regular expression case-insensitive. Choose the method that best fits your coding style and requirements.

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`). By default, it matches any character in the input string, except for the newline character.

Example:

import re

text = "Hello\nWorld"

# Regular expression pattern with a dot

pattern = re.compile(r'.')

matches = pattern.findall(text)

print(matches)

# Output: ['H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd']

In this example, the regular expression pattern `.` matches each individual character in the `text` except for the newline character, resulting in the output `['H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd']`.

However, if you pass `re.DOTALL` (or `re.S`) as the second argument when compiling the regular expression pattern, the `.` will also match newline characters.

Example:

import re

text = "Hello\nWorld"

# Regular expression pattern with DOTALL flag

pattern = re.compile(r'.', re.DOTALL)

matches = pattern.findall(text)

print(matches)

# Output: ['H', 'e', 'l', 'l', 'o', '\n', 'W', 'o', 'r', 'l', 'd']

In this example, the regular expression pattern `.` with `re.DOTALL` matches all characters in the `text`, including newline characters. The output includes `['H', 'e', 'l', 'l', 'o', '\n', 'W', 'o', 'r', 'l', 'd']`, where `\n` represents the newline character.So, passing `re.DOTALL` as the second argument in `re.compile()` changes the behavior of the dot `.` to match all characters, including newlines, in the input string.

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+')`, then `numReg.sub('X', '11 drummers, 10 pipers, five rings, 4 hens')` will return a new string where all occurrences of one or more digits are replaced by the letter 'X'.

Let's break down the process:

Input string: `'11 drummers, 10 pipers, five rings, 4 hens'`

A. The regular expression `r'\d+'` matches one or more consecutive digits in the input string.

B. The `sub()` method of the compiled regular expression `numReg` replaces all matched substrings with the letter 'X'.

Resulting output: `'X drummers, X pipers, five rings, X hens'`

In the output, all occurrences of digits '11', '10', and '4' have been replaced with the letter 'X', leaving the other parts of the input string unchanged.

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 more readable and organized regular expressions by enabling verbose mode. In verbose mode, you can include comments and whitespace within the regular expression pattern to make it easier to understand and maintain.

The `re.VERBOSE` flag is also known as the "verbose" or "extended" flag. When you use it, it allows you to create multi-line regular expressions with comments, extra whitespace, and line breaks, without affecting the actual matching behavior of the pattern. The Python interpreter will ignore comments and whitespace within the pattern when using `re.VERBOSE`.

Here's an example to illustrate how `re.VERBOSE` makes a difference:

import re

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

# Regular expression without VERBOSE flag

pattern\_without\_verbose = re.compile(r'The\w+brown', re.IGNORECASE)

# Regular expression with VERBOSE flag

pattern\_with\_verbose = re.compile(r'''

The # Match 'The'

\w+ # Match one or more word characters

brown # Match 'brown'

''', re.IGNORECASE | re.VERBOSE)

# Find matches using both patterns

matches\_without\_verbose = pattern\_without\_verbose.findall(text)

matches\_with\_verbose = pattern\_with\_verbose.findall(text)

print("Without VERBOSE:", matches\_without\_verbose)

print("With VERBOSE:", matches\_with\_verbose)

Output:

Without VERBOSE: ['The brown']

With VERBOSE: ['The brown']

In this example, both regular expressions will match the same part of the input string ("The brown"), but the one with `re.VERBOSE` is much easier to read and understand. The `#` symbol is used for comments, and whitespace and line breaks are added for better formatting.

The `re.VERBOSE` flag is particularly useful when you have complex regular expressions that require explanation or documentation. It helps you maintain a clear and organized pattern, making your code more readable and maintainable. However, it's important to note that `re.VERBOSE` only affects the readability of the pattern; it doesn't change the way the regular expression matches the text.

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 (like '1,234', '6,368,745'), you can use the following regular expression:

import re

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

strings\_to\_match = ['42', '1,234', '6,368,745']

strings\_to\_exclude = ['12,34,567', '1234']

for string in strings\_to\_match:

if re.match(pattern, string):

print(f"'{string}' matches the pattern.")

else:

print(f"'{string}' does not match the pattern.")

for string in strings\_to\_exclude:

if re.match(pattern, string):

print(f"'{string}' matches the pattern.")

else:

print(f"'{string}' does not match the pattern.")

Output:

'42' matches the pattern.

'1,234' matches the pattern.

'6,368,745' matches the pattern.

'12,34,567' does not match the pattern.

'1234' does not match the pattern.

Explanation of the regular expression `r'^\d{1,3}(,\d{3})\*$'`:

- `^`: Start of the string.

- `\d{1,3}`: Match one to three digits.

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

- `$`: End of the string.

The regular expression ensures that the number has at least one to three digits at the beginning and allows for zero or more repetitions of a comma followed by exactly three digits. This pattern enforces the requirement of having commas for every three digits in the number.

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' and whose first name is a single word starting with a capital letter, you can use the following regular expression:

import re

pattern = r'^[A-Z][a-zA-Z]\* Watanabe$'

names\_to\_match = ['Haruto Watanabe', 'Alice Watanabe', 'RoboCop Watanabe']

names\_to\_exclude = ['haruto Watanabe', 'Mr. Watanabe', 'Watanabe', 'Haruto watanabe']

for name in names\_to\_match:

if re.match(pattern, name):

print(f"'{name}' matches the pattern.")

else:

print(f"'{name}' does not match the pattern.")

for name in names\_to\_exclude:

if re.match(pattern, name):

print(f"'{name}' matches the pattern.")

else:

print(f"'{name}' does not match the pattern.")

Output:

'Haruto Watanabe' matches the pattern.

'Alice Watanabe' matches the pattern.

'RoboCop Watanabe' matches the pattern.

'haruto Watanabe' does not match the pattern.

'Mr. Watanabe' does not match the pattern.

'Watanabe' does not match the pattern.

'Haruto watanabe' does not match the pattern.

Explanation of the regular expression `r'^[A-Z][a-zA-Z]\* Watanabe$'`:

- `^`: Start of the string.

- `[A-Z]`: Match a single uppercase letter (the first letter of the first name).

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

- ` `: Match a space character.

- `Watanabe`: Match the last name 'Watanabe'.

- `$`: End of the string.

The regular expression ensures that the name starts with a single uppercase letter, followed by zero or more lowercase or uppercase letters (the first name), a space, and the last name 'Watanabe'. It does not match if the name is not properly capitalized or if there are extra characters between the first name and the last name.

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 sentences 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 case-insensitive regular expression:

import re

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

sentences\_to\_match = [

'Alice eats apples.',

'Bob pets cats.',

'Carol throws baseballs.',

'Alice throws Apples.',

'BOB EATS CATS.'

]

sentences\_to\_exclude = [

'RoboCop eats apples.',

'ALICE THROWS FOOTBALLS.',

'Carol eats 7 cats.'

]

for sentence in sentences\_to\_match:

if re.match(pattern, sentence, re.IGNORECASE):

print(f"'{sentence}' matches the pattern.")

else:

print(f"'{sentence}' does not match the pattern.")

for sentence in sentences\_to\_exclude:

if re.match(pattern, sentence, re.IGNORECASE):

print(f"'{sentence}' matches the pattern.")

else:

print(f"'{sentence}' does not match the pattern.")

Output:

'Alice eats apples.' matches the pattern.

'Bob pets cats.' matches the pattern.

'Carol throws baseballs.' matches the pattern.

'Alice throws Apples.' matches the pattern.

'BOB EATS CATS.' matches the pattern.

'RoboCop eats apples.' does not match the pattern.

'ALICE THROWS FOOTBALLS.' does not match the pattern.

'Carol eats 7 cats.' does not match the pattern.

Explanation of the regular expression `r'^(Alice|Bob|Carol) (eats|pets|throws) (apples|cats|baseballs)\.$'`:

- `^`: Start of the string.

- `(Alice|Bob|Carol)`: Match either 'Alice', 'Bob', or 'Carol' as the first word.

- ` `: Match a space character.

- `(eats|pets|throws)`: Match either 'eats', 'pets', or 'throws' as the second word.

- ` `: Match a space character.

- `(apples|cats|baseballs)`: Match either 'apples', 'cats', or 'baseballs' as the third word.

- `\.`: Match a period at the end of the string.

- `$`: End of the string.

The regular expression ensures that the sentence follows the specified pattern. It matches the provided sentences in both lowercase and uppercase due to the `re.IGNORECASE` flag. It will not match sentences that do not strictly follow this pattern or have other variations.