Q1. Explain the difference between greedy and non-greedy syntax with visual terms in as few words as possible. What is the bare minimum effort required to transform a greedy pattern into a non-greedy one? What characters or characters can you introduce or change?

Sol:-

The difference between greedy and non-greedy syntax in regular expressions is as follows:

Greedy syntax: It matches as much as possible, consuming the maximum number of characters that still allows the overall pattern to match. It is denoted by the \*, +, or ? modifiers.

Non-greedy syntax: It matches as little as possible, consuming the minimum number of characters needed for the overall pattern to match. It is denoted by adding ? after the \*, +, or ? modifiers.

To transform a greedy pattern into a non-greedy one, you simply add a ? after the \*, +, or ? modifier in the pattern. This minimal effort of adding a single ? changes the behavior from greedy to non-greedy.

For example, consider the following pattern:

Greedy pattern: .\*

Non-greedy pattern: .\*?

Q2. When exactly does greedy versus non-greedy make a difference?  What if you're looking for a non-greedy match but the only one available is greedy?

Sol:-

Greedy versus non-greedy matching makes a difference when there are multiple possible matches within a given input string.

In cases where there is only one possible match available and it is greedy, the distinction between greedy and non-greedy doesn't matter because there are no other alternatives to consider.

However, when there are multiple possible matches, the distinction becomes significant. Greedy matching tries to consume as much of the input as possible, often resulting in a longer match. Non-greedy matching, on the other hand, tries to consume as little as possible, resulting in a shorter match.

If you specifically need a non-greedy match but only a greedy match is available, you can modify the greedy pattern to become non-greedy by adding the ? modifier. This will change the behavior of the pattern and make it non-greedy. By making this modification, you can obtain the desired non-greedy match even if it was not initially available.

Q3. In a simple match of a string, which looks only for one match and does not do any replacement, is the use of a nontagged group likely to make any practical difference?

Sol:-

In a simple match of a string where you are not capturing any specific groups or using the matched groups in any further operations, the use of a non-tagged group (a group without a capturing group identifier) does not make a practical difference.

Non-tagged groups are typically used when you want to group a pattern for logical purposes but do not need to extract the matched content separately. These groups do not create a captured group that can be accessed later in the match object or used in replacement operations. If you are not utilizing captured groups in any way, using a non-tagged group versus not using any grouping at all would have no practical impact on the outcome of the match.

Q4. Describe a scenario in which using a nontagged category would have a significant impact on the program's outcomes.

Sol:-

The re.findall() function returns a list of all non-overlapping matches of a pattern in a string. When using capturing groups in the pattern, re.findall() returns a list of tuples where each tuple represents the matched groups. However, if you want to exclude certain groups from being captured and included in the result, you can use non-tagged groups.

Q5. Unlike a normal regex pattern, a look-ahead condition does not consume the characters it examines. Describe a situation in which this could make a difference in the results of your programme.

Sol:-

A situation where the non-consumptive nature of look-ahead conditions in regular expressions could make a difference in the results of your program is when you need to match a specific pattern only if it is followed by another specific pattern, without including the second pattern in the match.

import re

text = "Email: user@example.com, Domain: example.com"

pattern = r"\w+@(?=example\.com)"

matches = re.findall(pattern, text)

print(matches)

Q6. In standard expressions, what is the difference between positive look-ahead and negative look-ahead?

Sol:-

Positive look-ahead ((?=...)): It asserts that the pattern inside the look-ahead must be followed by the specified condition. It matches the pattern only if the condition is true. Positive look-ahead does not consume any characters in the string.

Negative look-ahead ((?!...)): It asserts that the pattern inside the look-ahead must not be followed by the specified condition. It matches the pattern only if the condition is false. Negative look-ahead does not consume any characters in the string.

To illustrate the difference, let's consider an example:

python

Copy code

import re

text = "apple orange banana"

# Positive look-ahead

positive\_pattern = r"\w+(?= orange)"

positive\_matches = re.findall(positive\_pattern, text)

print(positive\_matches) # Output: ['apple']

# Negative look-ahead

negative\_pattern = r"\w+(?! orange)"

negative\_matches = re.findall(negative\_pattern, text)

print(negative\_matches) # Output: ['apple', 'banana']

Q7. What is the benefit of referring to groups by name rather than by number in a standard expression?

Sol:-

Improved Readability: Using named groups makes the regular expression more self-explanatory and easier to understand, especially when dealing with complex patterns. Instead of relying on group indices, you can use meaningful names that describe the purpose of each group.

Code Maintainability: Named groups provide better code maintainability. If the regular expression pattern changes and the group order or count is modified, referencing groups by name ensures that the code remains valid and unaffected by the changes.

Self-Documenting Patterns: By using named groups, the regular expression pattern itself becomes more descriptive and self-documenting. The group names can provide additional context and clarify the intent of each group, making it easier for others to comprehend and modify the pattern if needed.

Accessing Group Results: When using named groups, the resulting match object provides a convenient way to access the captured groups using their names as keys. This eliminates the need to remember or track group indices when extracting specific information from the match.

Q8. Can you identify repeated items within a target string using named groups, as in "The cow jumped over the moon"?

Sol:-

Yes, you can use named groups in regular expressions to identify repeated items within a target string.

Here's an example that demonstrates how to use named groups to identify repeated words in a sentence:

import re

pattern = r'\b(?P<word>\w+)\b(?P=word)'

sentence = "The cow jumped over the moon"

matches = re.findall(pattern, sentence)

repeated\_words = set(matches)

print(repeated\_words) # Output: {'the'}

Q9. When parsing a string, what is at least one thing that the Scanner interface does for you that the re.findall feature does not?

Sol:-

Incremental tokenization: The Scanner interface allows you to tokenize the input string incrementally. It maintains a state between successive calls to the scan() method, which enables you to process the string token by token rather than extracting all matches at once.

Pattern-based matching: With the Scanner interface, you can define different patterns to match specific tokens or patterns in the input string. This gives you more control over the parsing process and allows you to handle different token types or structures in a more structured manner.

Skipping unwanted characters: The Scanner interface allows you to define patterns for skipping or ignoring certain characters or patterns in the input string. This is particularly useful when you want to exclude certain elements from the parsing process, such as whitespace or comments.

Q10. Does a scanner object have to be named scanner?

Sol:-

No, a Scanner object does not have to be named "scanner." The name of the Scanner object is arbitrary and can be chosen based on your preference or the context of your code. You can use any valid variable name that adheres to Python's naming conventions.