**ADITYA AMIN**

**ASSIGN : 17**

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?

Greedy: Matches as much as possible, trying to consume the longest substring that satisfies the pattern.

Visual representation: "Grabbing" the largest possible chunk.

Non-greedy (also called lazy or reluctant): Matches as little as possible, trying to consume the shortest substring that satisfies the pattern.

Visual representation: "Releasing" the smallest possible chunk.

To transform a greedy pattern into a non-greedy one, the minimum effort required is to introduce a question mark (?) immediately after the quantifier (\*, +, ?, or {m,n}). This makes the quantifier non-greedy. Alternatively, you can replace the quantifier with its non-greedy counterpart (\*?, +?, ??, or {m,n}?).

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?

The difference between greedy and non-greedy matching in regular expressions becomes relevant when there are multiple possible matches for a given pattern. It determines the extent to which the pattern tries to consume characters in order to find a match.

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?

In a simple match scenario where you are not capturing or using the groups for further processing, the use of a non-tagged group is not likely to have any practical difference in terms of the match result itself.

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

A scenario where using a non-tagged category (non-capturing group) can have a significant impact on the program's outcomes is when you are using the re.findall() function in Python's regular expressions module (re).

The re.findall() function is used to find all non-overlapping occurrences of a pattern in a string and return them as a list of strings. By default, when capturing groups are present in the pattern, re.findall() returns a list of tuples, where each tuple represents a match and contains the captured 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.

Consider the following scenario: You have a text document that contains lines of text, and you want to find all lines that contain the word "apple" followed by the word "pie" on the same line. However, you don't want to include the word "apple" in the match result.

Using a look-ahead condition in this scenario allows you to specify the constraint without consuming the characters. You can create a pattern that matches "apple" only if it is followed by "pie" without including "apple" in the final match result.

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

In both cases, the look-ahead condition is used to specify additional constraints for the pattern match. Positive look-ahead ensures the presence of a specific pattern, while negative look-ahead ensures the absence of a specific pattern.

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

Improved Readability: Using named groups makes the regular expression more readable and self-explanatory. By assigning meaningful names to groups, you can convey the purpose and intent of each group in the pattern, making it easier to understand and maintain the regular expression.

Better Code Maintainability: Named groups enhance the maintainability of the code. If the regular expression needs to be modified or extended in the future, using named groups makes it easier to identify and update specific groups without relying on their numerical indices. This reduces the chances of introducing errors while modifying the regular expression.

Clearer Access to Matched Substrings: When a regular expression matches a string, accessing the matched substrings by their named groups provides a clearer and more descriptive way to extract the desired information.

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

Yes, you can use named groups in regular expressions to identify repeated items within a target string. However, it's important to note that named groups themselves do not inherently detect repetitions. Instead, they allow you to capture and refer to specific parts of a matched substring.

import re

text = "The cow jumped over the moon"

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

matches = re.findall(pattern, text)

print(matches) # 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?

Stateful Parsing: The Scanner interface maintains a state while iterating over the string. It remembers the position where it left off, allowing you to resume parsing from that point. This can be useful when dealing with complex string parsing scenarios where you need to handle different contexts or maintain parsing state across multiple iterations.

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

No, a Scanner object does not have to be named "scanner." The name given to a Scanner object is arbitrary and can be chosen based on your preference or the context of your code.