**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 syntax in regular expressions matches the longest possible string that satisfies the pattern, while non-greedy (also known as lazy or reluctant) syntax matches the shortest possible string that satisfies the pattern.

To transform a greedy pattern into a non-greedy one, you can introduce the? quantifier after a greedy quantifier (\*, +, ?, {m,n}) or change a greedy quantifier to its non-greedy counterpart (\*?, +?, ??, {m,n}?).

For example:

* .\* (greedy) matches as much as possible.
* .\*? (non-greedy) matches as little as possible.

**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 Greedy Match will try to match as many repetitions of the quantified pattern as possible. The Non Greedy Match will try to match as few repetitions of the quantified pattern as possible. If only Non Greedy Match is available, we can use other filtering or pattern matching methods of regex and further identify the required pattern.

**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 non-tagged group likely to make any practical difference?**

In a simple match of a string where no replacement or capturing of groups is involved, the use of a non-tagged group (a group without capturing parentheses) generally does not make any practical difference in terms of the match result.

Non-tagged groups in regular expressions are used when you need to apply a specific pattern to a portion of the string but do not require capturing that portion for further use. These groups are primarily used for grouping and applying quantifiers to a sequence of characters.

import re

pattern = r'ab(?:cd)\*ef'

text = 'abcdcdcdef'

# Matching with non-tagged group

match\_obj = re.match(pattern, text)

if match\_obj:

matched\_string = match\_obj.group()

print(matched\_string) # Output: 'abcdcdcdef'

n this example, the non-tagged group (?:cd)\* is used to match the sequence "cd" zero or more times. It helps in applying the quantifier \* to the pattern "cd" without capturing it as a group. The resulting match is the same as if we had used a tagged group '(cd)\*'.

**Q4. Describe a scenario in which using a non-tagged category would have a significant impact on the program's outcomes.**

Here in the below Code Snippet . decimal is not tagged or captured. It will useful in scenarios where the separator of value in a string is of no use and we need to capture only the values.

import re

text='135.456'

pattern=r'(\d+)(?:.)(\d+)'

regobj=re.compile(pattern)

matobj=regobj.search(text)

matobj.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.**

While counting the number of multiple lines or multiple sentence in a string the positive look ahead makes a difference, without which we won’t get the correct count of lines or sentences in a string.

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

Positive Look-ahead allows to add a condition for what follows. Negative Look-ahead is similar, but it looks behind. That is, it allows to match a pattern only if there’s something before it.

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

Referring to groups by name rather than by number in a standard expression helps to keep the code clear and easy to understand.

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

import re

text = "The cow jumped over the moon"

regobj=re.compile(r'(?P<w1>The)',re.I)

regobj.findall(text)

**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?**

re.findall () module is used to search for all occurrences that match a given pattern. In contrast, re.search () will only return the first occurrence that matches the specified pattern. re.findall () will iterate over all the lines of the file and will return all non-overlapping matches of pattern in a single step.

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

Yes, it may have any name.