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?

***Ans***:

In regular expressions,

Greedy syntax matches as much text as possible, while non-greedy syntax matches as little text as possible. Greedy syntax is denoted by adding a + or \* to a pattern, while non-greedy syntax is denoted by adding a ‘?’ to a pattern.

To transform a greedy pattern into a non-greedy one, you can simply add a ‘?’ after the ‘+’ or ‘\*’ quantifier. For example, the greedy pattern ‘.\*’ can be transformed into a non-greedy pattern ‘.\*?’ by adding a ‘?’ after the ‘\*’.’

The ‘?’ character can also be used to make other quantifiers non-greedy. For example, the pattern ‘a{2,5}’ matches between 2 and 5 consecutive ‘a’ characters, and is greedy by default. To make it non-greedy, you can add a ‘?’ after the quantifier, like this: ‘a{2,5}?’.

Visually, the difference between greedy and non-greedy syntax can be represented by thinking of the regex engine as a "consumption machine". Greedy syntax causes the regex engine to consume as much of the input string as possible, while non-greedy syntax causes it to consume 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?

***Ans***:

Greedy versus non-greedy syntax makes a difference in regular expressions when the pattern contains a quantifier, such as \*, +, or ?, that can match multiple occurrences of a character or group.

If you're looking for a non-greedy match but the only one available is greedy, you can try using a negative lookahead assertion to exclude the characters that you don't want to 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?

***Ans***:

In a simple match of a string where you are only looking for one match and not doing any replacement, using a non-tagged group is not likely to make any practical difference.

Non-tagged groups are typically used for capturing substrings within a larger string, and if you are not capturing any substrings, there is no need to use a non-tagged group.

However, there are certain situations where using a non-tagged group may be useful, such as when you want to group together multiple elements in a regular expression for readability or to apply quantifiers to them.

In general, though, if you are only looking for a single match and not capturing any substrings, using a non-tagged group is not likely to make a practical difference.

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

***Ans***:

(?:http[s]?)://(\S+)

The regular expression above uses a non-tagged category to group the "http" or "https" part of the URL.

This means that this part of the URL won't be captured in the result of the regular expression match.

By using a non-tagged category, you can ensure that the regular expression matches all URLs, regardless of whether they start with "http" or "https".

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.

***Ans***:

If you use the regex pattern (\w+@\w+\.com)(?=.\*gmail), which matches any email address followed by "gmail" using a positive look-ahead condition, but without including a negative look-behind condition to exclude email addresses preceded by "fake", your program will match all four email addresses in the string, including "[fakejohndoe@gmail.com](mailto:fakejohndoe@gmail.com)" and "[fakemail@gmail.com](mailto:fakemail@gmail.com)", which are not valid.

To avoid this problem, you can modify the regex pattern to include a negative look-behind condition to exclude email addresses preceded by "fake". For example, you can use the regex pattern (?<!fake)(\w+@\w+\.com)(?=.\*gmail) which matches any email address followed by "gmail" but not preceded by "fake". This will correctly match only "[johndoe@gmail.com](mailto:johndoe@gmail.com)" in the above string.

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

***Ans***:

Positive look-ahead assertions are used to match a pattern only if it is followed by another pattern, while Negative look-ahead assertions are used to match a pattern only if it is not followed by another pattern.

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

***Ans***: Using named groups in regular expressions can make your code more readable, maintainable, and less prone to errors when the order or number of groups changes.

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

***Ans***: Yes, you can use named groups to identify repeated items within a target string in a regular expression.

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?

***Ans***: the Scanner interface provides additional functionality for tokenizing input and parsing different types of data, which can be useful in certain situations.

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

***Ans***: No, a Scanner object does not have to be named scanner.