**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- Greedy syntax tries to match the longest possible string that fits a given pattern, while non-greedy syntax matches the shortest possible string.

To transform a greedy pattern into a non-greedy one, you can introduce the question mark '?' after the quantifier (such as '\*', '+', or '{n,m}') to make it lazy.

For example, the greedy pattern "." would match the entire string "hello world, how are you?", while the non-greedy pattern ".?" would match only "hello".

**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 when there are multiple possible matches for a given pattern. In greedy syntax, the engine will try to match the longest possible substring that satisfies the pattern, while in non-greedy syntax, the engine will try to match the shortest possible substring that satisfies the pattern.

If you are looking for a non-greedy match but the only one available is greedy, you can try to modify the pattern to make it non-greedy by adding the question mark '?' after the quantifier. However, if that is not possible, you may need to use additional logic or manipulation of the matched substring to extract the desired non-greedy match. For example, you can use a negative lookahead assertion to exclude certain substrings from the match, or use a capturing group to extract only a portion of the 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, using a non-tagged group is not likely to make a practical difference if you are only interested in finding one match and not capturing any groups. Non-capturing groups (groups that start with ?:) are used to group together a sequence of characters without capturing them as a separate match. They can be useful in more complex regular expressions where you need to use the group later on in the expression or to improve readability.

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

Ans- Non-tagged categories, also known as non-capturing groups, are used in regular expressions to group together a sequence of characters without capturing them as a separate match. While using a non-capturing group may not always have a significant impact on the outcome of a program, there are certain scenarios where it can make a significant difference. One such scenario is when working with large datasets and complex regular expressions.

Consider a scenario where you have a large dataset containing email addresses, and you need to extract the domain name from each email address. You could use a regular expression such as ‘@([a-z]+\.[a-z]+)’ to match the domain name, where the group ‘([a-z]+\.[a-z]+)’ captures the domain name. However, if you have a large dataset with millions of email addresses, capturing each domain name as a separate match can consume a lot of memory and slow down your program.

**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- Look-ahead assertions in regular expressions are used to specify a condition that must be met in order for a match to occur, without consuming the characters that make up the condition. One situation where this can make a difference in the results of your program is when you need to match a pattern that is followed by a certain condition, but you do not want to include the condition in the matched text.

Example: [\w.+-][+@(?=.\*example\.com).\*](mailto:+@(?=.*example\.com).*)

In this regular expression, the look-ahead assertion (?=.\*example\.com) specifies that the email address must be followed by the string "example.com", but it does not consume the characters that make up the "example.com" domain. The .\* at the end of the regular expression matches any remaining characters in the email address, but because the look-ahead assertion does not consume any characters, only the part of the email address before the "example.com" domain is included in the matched text.

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

Ans-In regular expressions, look-ahead is a type of assertion that allows you to match a pattern only if it is followed by another pattern. Look-ahead assertions are divided into two types: positive and negative. Positive look-ahead is used to match a pattern only if it is followed by another pattern, while negative look-ahead is used to match a pattern only if it is not followed by another pattern.

Positive look-ahead is denoted by ‘(?=...)’ and it matches the search pattern only if it is followed by the positive lookahead pattern. The positive lookahead pattern is not included in the matched string. For example, the regular expression ‘a(?=b)’ matches the 'a' character only if it is followed by a 'b' character, but it does not include the 'b' character in the matched string.

Negative look-ahead, on the other hand, is denoted by (?!...) and matches the search pattern only if it is not followed by the negative lookahead pattern. Like positive lookahead, the negative lookahead pattern is also not included in the matched string. For example, the regular expression a(?!b) matches the 'a' character only if it is not followed by a 'b' character.

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

Ans-Referring to groups by name rather than by number in a regular expression can make the code more readable, maintainable, and less prone to errors. It provides some benefits like Clarity, Readability, Flexibility and Self-documenting.

**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 identify repeated items within a target string using named groups in regular expressions.

For example, if you want to identify any repeated words in the target string "The cow jumped over the moon", you could use a regular expression with a named group.

**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 and the re.findall() function serve different purposes and have different capabilities when it comes to parsing strings. The Scanner interface is part of the Java API and is used for parsing text data from various input sources, such as files or streams. It provides methods for reading and parsing data based on a specified delimiter pattern, as well as methods for retrieving the parsed data in various formats. One advantage of the Scanner interface over the re.findall() function is that it can handle more complex parsing tasks, such as parsing different data types (integers, floating-point numbers, etc.), skipping over whitespace and comments, and handling different character encodings.

On the other hand, the ‘re.findall()’ function is part of the Python re module and is primarily used for pattern matching and extraction within strings. It returns a list of all non-overlapping matches of a regular expression within a string, but does not provide any built-in support for parsing data types or handling more complex parsing tasks.

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

Ans- No, a ‘Scanner’ object does not have to be named ‘scanner’. You can name it whatever you want, as long as the name is a valid Java identifier and is not already used for another variable or object in your code.