**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?**

**ANSWER:** In regular expressions, greedy syntax matches the longest possible string that satisfies the pattern, while non-greedy syntax matches the shortest possible string that satisfies the pattern.

To transform a greedy pattern into a non-greedy one, you can add a question mark after the quantifier. For example, changing .\* to .\*? makes it non-greedy.

Adding the question mark turns the greedy quantifier (matching the longest possible string) into a non-greedy quantifier (matching the shortest possible string). Other quantifiers, such as + and ?, can also be modified in the same way to switch between greedy and non-greedy matching.

**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?**

**ANSWER:** The difference between greedy and non-greedy matching can make a difference when there are multiple possible matches in a string that satisfy the same pattern.

In greedy matching, the engine will try to match as much of the string as possible while still satisfying the pattern. This can lead to unexpected results if there are multiple possible matches, as the engine may match more than intended.

In non-greedy matching, the engine will try to match as little of the string as possible while still satisfying the pattern. This can be useful when you want to match the smallest possible substring that satisfies a pattern.

If you are looking for a non-greedy match but the only one available is greedy, you may need to modify your pattern or use a different approach. Depending on the specifics of your use case, you might be able to use a negative lookahead or lookbehind to exclude certain substrings from the match. Alternatively, you could use a different pattern that is more specific to the data you are working with.

**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?**

**ANSWER:** In a simple match of a string that looks only for one match and does not do any replacement, the use of a non-capturing group (also known as a nontagged group) is not likely to make any practical difference.

Non-capturing groups are typically used to group together parts of a pattern without capturing the matched substring. This can be useful for more complex patterns or when using quantifiers that apply to multiple elements in the pattern.

However, in a simple match where only one match is expected and no captured groups are needed, the use of a non-capturing group is not necessary. The use of capturing groups, on the other hand, can impact the match result by capturing and returning the matched substring.

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

**ANSWER:** One scenario where using a non-capturing group (also known as a nontagged group) can have a significant impact on the program's outcomes is when working with regular expressions that use quantifiers.

Quantifiers such as \*, +, and ? apply to the previous element in the pattern, and can match multiple occurrences of that element. When using quantifiers, a non-capturing group can be used to group together multiple elements and apply the quantifier to the group as a whole.

For example, consider a regular expression that matches phone numbers in a specific format, such as (555)-555-5555. To match this pattern with a varying number of spaces or dashes between the digits, you might use a pattern like (\d\s\*-\*\s\*){3}\d.

**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.**

**ANSWER:** A situation where the non-consumptive nature of a lookahead condition in a regex pattern could make a difference in the results of a program is when working with overlapping matches.

Consider a simple example where we want to match all occurrences of the substring "cat" in a given string. If we use a pattern like cat, the regex engine will find and match all instances of "cat" in the string, consuming the characters that make up each match.

However, if we want to find all occurrences of "cat" that are followed by the substring "dog", we might use a lookahead condition like cat(?=dog). This pattern matches all occurrences of "cat" that are immediately followed by "dog", but it does not consume the characters that make up the "dog" substring.

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

**ANSWER:** In standard regular expressions, positive lookahead and negative lookahead are two types of lookahead assertions that allow you to specify a condition that must (or must not) be true immediately after a certain point in the string.

A positive lookahead assertion is denoted by (?=...), and it matches a substring if the pattern inside the lookahead matches the substring immediately after the current position in the string. Positive lookahead does not consume the characters that make up the matched substring.

A negative lookahead assertion is denoted by (?!...), and it matches a substring if the pattern inside the lookahead does not match the substring immediately after the current position in the string. Negative lookahead does not consume the characters that make up the matched substring.

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

**ANSWER:** Referring to groups by name rather than by number in a standard regular expression can make the pattern more readable and easier to maintain, especially when dealing with complex patterns that involve multiple groups.

When you refer to groups by name, you can use descriptive names that make the purpose of each group clear. Using named groups also makes the pattern more robust to changes in the order or number of the captured groups

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

**ANSWER:** Named groups in regular expressions can be used to capture and extract repeated items within a target string. However, in the example you provided ("The cow jumped over the moon"), there are no repeated items to capture.

**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?**

**ANSWER:** One thing that the Scanner interface does for you that re.findall does not is it allows you to read and parse the input string in a streaming fashion. This means that you can process the input string one token at a time, rather than reading the entire string into memory at once. This can be useful for processing large input strings that would otherwise cause memory issues.

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

**ANSWER:** No, a scanner object in Python does not have to be named "scanner". In fact, there is no built-in Scanner class in Python. However, you can use regular expressions in Python's re module to achieve similar functionality as a scanner.