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?

In regular expressions, greedy syntax matches the longest possible string that satisfies the pattern, while non-greedy or lazy syntax matches the shortest possible string. To transform a greedy pattern into a non-greedy one, add a question mark (?) to the end of the quantifier, such as '\*?' or '+?'.

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?

Greedy versus non-greedy makes a difference when the target string has multiple matches that satisfy the pattern, and the goal is to find the shortest or longest match. If the only available match is greedy, it may need to be refined by using a non-greedy approach to match only the desired part of the string.

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 of a string, using a nontagged group may not make a practical difference since it does not affect the matching result or the captured groups. However, using a named group can make the code more readable and easier to maintain.

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 nontagged category would have a significant impact is when you want to match a specific sequence of characters but exclude certain characters that are not part of the match. For example, if you want to match all words that start with "a" and end with "c", but exclude any words that contain the letter "b", you can use a negative lookahead category: r'\ba(?!.\*b).\*c\b'.

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 program.

In a situation where you need to match a pattern that occurs in a specific context, using a look-ahead condition can ensure that the context is correct without including it in the match. For example, if you want to match a phone number that follows the pattern (XXX) XXX-XXXX, you can use a positive look-ahead assertion to match the closing parenthesis without including it in the match: r'\d{3}(?=)).\*\d{3}-\d{4}'.

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

In standard expressions, a positive look-ahead assertion matches a group of characters if the group is followed by another group of characters that match a pattern, while a negative look-ahead assertion matches a group of characters if the group is not followed by another group of characters that match a pattern.

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 makes the code more readable and easier to maintain, especially when the pattern contains multiple groups. It also allows for easier access to the groups in the code, using the group name as a dictionary key.

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 to identify repeated items within a target string. For example, in the string "The cow jumped over the moon", you could use the regular expression pattern **(?P<word>\w+)\s+(?P<word>\w+)\s+(?P<word>\w+)\s+(?P<word>\w+)\s+(?P<word>\w+)** to capture each word in a separate group. Note that the group names must be unique.

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?

One thing that the Scanner interface does for you when parsing a string that the re.findall feature does not do is that it allows you to specify the type of token that you want to match. The Scanner interface has built-in support for matching tokens such as integers, floats, and strings, which can be more convenient than using regular expressions to match these types of tokens.

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

No, a scanner object does not have to be named scanner. You can name a scanner object whatever you like, as long as the name is a valid Python identifier. However, it is good practice to choose a name that is descriptive of the object's purpose to make your code more readable and easier to understand.