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 as much as possible, while non-greedy syntax matches as little as possible.

To transform a greedy pattern into a non-greedy one, you can introduce or change the ? character immediately after a quantifier (\*, +, ?, {m,n}). This modifies the quantifier to be non-greedy, causing it to match the minimum number of occurrences.

For example, given the greedy pattern .\*, which matches any characters as much as possible, you can transform it into a non-greedy pattern by adding ? after the \*, resulting in .\*?. This makes the quantifier \* non-greedy, matching the minimum number of characters.

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 matching makes a difference when there are multiple possible matches that satisfy the given pattern.

In the case of a greedy match, the pattern will try to match as much as possible, potentially consuming more characters than intended. This can be a problem when you want to match specific portions of the input and avoid consuming more than necessary.

On the other hand, non-greedy matching is designed to match as little as possible while still satisfying the pattern. This is useful when you want to find the smallest possible match or when you need to extract specific portions of the input.

If you specifically need a non-greedy match but the only available option is a greedy match, you can modify the greedy pattern to be non-greedy by adding the ? character after the quantifier. This allows you to obtain the desired non-greedy behavior even if it is not the default option.

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 where you are only looking for one match and not performing any replacement, the use of a non-tagged group does not make a practical difference.

Non-tagged groups, denoted by parentheses without capturing group tags (such as (?:pattern)), are used when you want to group a pattern but don't need to capture the matched substring. They are primarily used for grouping and applying quantifiers to a specific pattern without creating a capture group.

In a simple match scenario where you are not capturing any groups or extracting specific substrings, the use of a non-tagged group is not necessary. It will not affect the outcome or behavior of the match.

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 non-tagged category (non-capturing group) can have a significant impact on program outcomes is when you are using regular expressions with alternations (the vertical bar |) and capturing groups to extract specific information.

For example, let's say you have a text document with lines containing names and phone numbers in different formats, such as:

John Doe: (123) 456-7890

Jane Smith: 987-654-3210

Mike Johnson: 555-1234

You want to extract the names and phone numbers separately. You can use a regular expression pattern like (\w+ \w+):\s(\d{3}-\d{3}-\d{4}|\(\d{3}\) \d{3}-\d{4}) to capture the name and phone number in separate groups.

If you use a capturing group around the alternation ( ... | ... ), it will create separate capture groups for each alternative, even if only one of them matches. This means that you will have to check which capture group has a non-empty value to determine if the phone number is in the format (\d{3}-\d{3}-\d{4}) or (\(\d{3}\) \d{3}-\d{4}).

However, if you use a non-tagged category (non-capturing group) around the alternation (?: ... | ... ), it will still match the correct phone number format, but it won't create separate capture groups. You will have only one capture group for the phone number, making it easier to extract the information without additional checks.

So, in this scenario, using a non-tagged category can have a significant impact on the program's outcomes by simplifying the extraction of specific information from the text.

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.

One situation where the non-consumption of characters by a look-ahead condition can make a difference in the results of a program is when you want to match a specific pattern that is followed by another pattern but you don't want the following pattern to be included in the final match.

For example, let's consider a scenario where you have a list of email addresses and you want to extract the usernames (part before the "@" symbol) that are followed by a specific domain name.

If you use a normal regex pattern without a look-ahead condition, such as (\w+)@example.com, it will match the entire email address (username@example.com) and capture only the username portion. However, the "@example.com" part will still be consumed by the pattern, meaning it will be part of the match.

On the other hand, if you use a look-ahead condition in your pattern, such as (\w+)(?=@example.com), the look-ahead (?=@example.com) ensures that the matching process only checks if the pattern @example.com follows the username, but it doesn't consume the characters of @example.com. As a result, only the username part will be captured in the match, excluding the domain name.

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

In regular expressions, positive look-ahead and negative look-ahead are two types of look-ahead assertions that allow you to specify conditions that must (positive) or must not (negative) be met for a match to occur.

Positive look-ahead is denoted by (?=...) syntax and specifies that the pattern inside the lookahead must be present after the current position in the string, but it does not consume any characters. In other words, it asserts that the pattern following the look-ahead should match, but it does not include those characters in the actual match.

Negative look-ahead is denoted by (?!...) syntax and specifies that the pattern inside the lookahead must not be present after the current position in the string. It is a negative assertion, meaning it asserts that the pattern following the look-ahead should not match.

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 regular expression provides several benefits:

1. Improved Readability: Naming groups in a regular expression makes it more readable and easier to understand the intent of each group. Using descriptive names for groups can make the regular expression code self-explanatory and reduce the need for comments.
2. Better Maintenance: When modifying a regular expression pattern, referencing groups by name makes it easier to understand the purpose of each group and update the pattern accordingly. It enhances the maintainability of the code by reducing the chances of introducing errors when modifying group references.

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

Yes, we can use named groups to identify repeated items within a target string. However, in the example you provided ("The cow jumped over the moon"), there are no repeated items.

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?

When parsing a string, the Scanner interface in Python provides a higher level of control and flexibility compared to the re.findall() function. One thing that the Scanner interface does for you that re.findall() does not is allowing you to define different token patterns and actions for each pattern.

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

No, a scanner object does not have to be named "scanner." The name of the scanner object can be any valid variable name that follows the naming rules in Python. The name "scanner" is commonly used as a convention, but you are free to choose any name that makes sense and is meaningful in the context of your code.