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?

A1. In regular expressions, greedy syntax matches the longest possible substring that satisfies the pattern, while non-greedy syntax matches the shortest possible substring. The bare minimum effort required to transform a greedy pattern into a non-greedy one is to add a question mark (?) after the repetition character (\* or +). This question mark changes the pattern to a non-greedy one. For example, the greedy pattern ".*" matches any character (.) zero or more times (*) as many times as possible, while the non-greedy pattern ".*?" matches any character (.) zero or more times (*) as few times 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?

A2.   
Greedy and non-greedy matching make a difference when there are multiple possible matches for a given pattern in a string.

In greedy matching, the pattern matches as many characters as possible while still allowing the overall pattern to match the entire string.

In non-greedy matching, the pattern matches as few characters as possible while still allowing the overall pattern to match the entire string.

For example, given the string "abcabcabc" and the pattern "ab.\*c", the greedy match would be "abcabcab" and the non-greedy match would be "abc".

If the only available match is greedy but you want a non-greedy match, you can add a "?" after the qualifier in the pattern. For example, the pattern "ab.\*?c" would perform a non-greedy match in the above example.

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?

A3. In a simple match of a string that only looks for one match and does not do any replacement, the use of a nontagged group is unlikely to make any practical difference.

Non-tagged groups are used to group multiple characters together for quantification or alternation purposes, but they do not capture or store the matched substring as a separate group that can be referenced later. Therefore, in a simple match that only seeks to find a match for a specific pattern, the presence or absence of a non-tagged group would not affect the result.

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

A4. In regular expressions, the use of a nontagged category, such as [^...], can have a significant impact on the program's outcomes in certain scenarios. One such scenario is when you need to match any character except a specific set of characters. For example, suppose you want to match a string that contains any character except vowels (a, e, i, o, u). You could use the following regular expression: [^aeiou]+. This pattern will match one or more of any character except vowels. If you use a tagged category, such as [a-z]+, you would need to add each vowel separately as an exclusion, which would make the pattern more complex and less efficient. Therefore, in this scenario, using a nontagged category can have a significant impact on the program's outcomes, making it easier to write a more efficient and concise regular expression.

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.

A5. A situation where a look-ahead condition could make a difference in the results of a program is when we need to match a pattern based on the presence or absence of a certain character without including that character in the actual match. For example, suppose we want to match all words that are followed by a colon, but we don't want to include the colon in the match. In this case, we can use a positive look-ahead condition to specify that the colon should be present, but not include it in the actual match. If we were to use a normal regex pattern without the look-ahead condition, we would end up including the colon in the match, which is not what we want.

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

A6.   
In regular expressions, look-ahead assertions are used to check if a pattern is followed by another pattern or not. Positive look-ahead assertions are those that match a group of characters that is followed by a specific pattern. Negative look-ahead assertions are those that match a group of characters that is not followed by a specific pattern.

The main difference between positive and negative look-ahead assertions is that positive look-ahead assertions ensure that the pattern following the matched pattern does exist, while negative look-ahead assertions ensure that the pattern following the matched pattern does not exist.

For example, the regular expression **foo(?=bar)** matches "foo" only if it is followed by "bar". On the other hand, the regular expression **foo(?!bar)** matches "foo" only if it is not followed by "bar".

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

A7.   
Referring to groups by name rather than by number in a standard expression has several benefits:

1. Clarity: Group names can make the regular expression more readable and self-explanatory. It can help other developers understand the code more easily, especially when dealing with complex regular expressions.
2. Flexibility: Using group names can make the code more flexible. For instance, if the order of the groups changes, the code will still be able to match the correct groups, as the groups are referred to by name rather than number.
3. Maintainability: If the regular expression is modified later on, using group names can make it easier to update the code, as the group names do not change even if the order or number of groups changes.
4. Accessibility: Referring to groups by name can be beneficial when working with non-technical people, who may find numbers harder to understand than descriptive names.

In summary, using group names can make the code more readable, flexible, maintainable, and accessible.

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

A8. Yes, you can identify repeated items within a target string using named groups in regular expressions. For example, if you want to match a word that appears twice in a row, you can use a named group as follows:

import re

pattern = r'(?P<word>\b\w+\b)(\s+\g<word>)+'

text = 'The cow jumped over the moon and the cow jumped over the fence'

matches = re.findall(pattern, text)

print(matches)

In this example, the regular expression pattern matches any word character (**\w**) that appears one or more times (**+**) between word boundaries (**\b**). This is captured in a named group called **word**. Then, the pattern looks for one or more whitespace characters (**\s+**) followed by the same word captured in the named group (**\g<word>**). This is repeated one or more times using the **+** quantifier.

When this regular expression is applied to the sample text, it matches the word "cow" twice because it appears consecutively in the text.

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?

A9. The Scanner interface is part of the Java language and is used to parse data from a string or input stream. One advantage of using the Scanner interface over the **re.findall** function in Python is that it allows for the parsing of different data types, such as integers, floats, and strings, by using specific methods such as **nextInt()**, **nextDouble()**, and **nextLine()**. The **re.findall** function in Python, on the other hand, returns a list of all non-overlapping matches in a string for a specified regular expression pattern.

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

A10. No, a **Scanner** object can be named anything that is a valid variable name in Python. The name of the object is simply a reference to the instance of the **Scanner** class, and it is up to the programmer to choose a name that is meaningful and easy to read.