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?

Solution - Greedy and non-greedy syntax in regular expressions determine the behavior of pattern matching when there are multiple possible matches.
Greedy: Matches as much as possible, consuming as many characters as it can.
Non-greedy (also known as lazy or reluctant): Matches as little as possible, consuming the fewest characters necessary.

To transform a greedy pattern into a non-greedy one, you can introduce or change the following character:
Add a question mark (?) after a quantifier (*, +, {}) in the pattern.
For example, let's consider the pattern .* which matches any characters (greedy).
To transform it into a non-greedy pattern, you can change it to .*?.
The addition of the question mark makes the quantifier non-greedy, causing it to match as few characters 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?

Solution - The distinction between greedy and non-greedy matching becomes relevant when there are multiple possible matches within the given input text.
Greedy Matching: In greedy matching, the regex engine tries to match as much as possible while still allowing the overall pattern to succeed. It consumes as many characters as it can.
Non-greedy Matching: In non-greedy (or lazy) matching, the regex engine tries to match as little as possible while still allowing the overall pattern to succeed. It consumes the fewest characters necessary.

If you are looking for a non-greedy match but the only available match is a greedy one, you can try using additional patterns or techniques to achieve the desired non-greedy behavior. Here are a few approaches you can consider:

Adjust the Pattern:
Modify the pattern to be more specific or use additional constraints to guide the non-greedy matching.
For example, instead of using a wildcard character like .*, consider using a more explicit pattern that matches a specific set of characters or a specific pattern.

Use Lookahead:
Utilize lookahead assertions to ensure that the match stops at a certain point.
Lookahead allows you to check if a certain pattern follows without including it in the actual match.
By incorporating lookahead in your pattern, you can control the stopping point of the match.

Split and Process:
If the input text contains multiple occurrences of the desired match, but the greedy pattern matches all of them as one, you can split the input text into separate substrings and apply the pattern to each substring individually.
This way, you can process each non-greedy match separately.

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?

Solution - In a simple match of a string where you are only interested in finding a single match and not performing any replacements or capturing specific groups, the use of a non-tagged group in the regular expression is unlikely to make any practical difference.

In regular expressions, groups are typically used for capturing specific portions of the matched string or for creating subexpressions to apply quantifiers or alternations. Non-tagged groups, also known as non-capturing groups, are denoted by (?:...) syntax, where the ?: indicates that the group will not capture the matched substring as a separate group.
When you are only interested in the overall match and do not need to capture any specific portions of the matched string, using a non-tagged group or omitting the use of groups altogether should not impact the practical outcome of the match.

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

Solution - One scenario where using a non-tagged category (non-capturing group) can have a significant impact on the program's outcomes is when you're utilizing backreferences in your regular expression pattern. Backreferences allow you to refer to previously matched groups within the pattern. When using backreferences, the presence or absence of capturing groups can affect the behavior and results of the pattern.

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.

Solution - The non-consumptive nature of a look-ahead condition in regular expressions can make a difference in the results of a program when you want to apply a pattern based on a specific condition without including the condition itself in the matched result.
Here's an example scenario:
Suppose you have a list of filenames, and you want to match only the filenames that have the extension ".txt" but exclude those that contain the word "important" before the extension.

files = ['document.txt', 'important_document.txt', 'notes.txt', 'important_notes.txt']

If you use a regular expression without a look-ahead condition, such as .*(?<!important)\.txt, the negative lookbehind (?<!important) ensures that the word "important" does not appear before the extension ".txt". However, this pattern would also consume the characters before the extension, resulting in the entire filename being included in the match.

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

Solution - In regular expressions, positive look-ahead and negative look-ahead are two types of look-around assertions that allow you to specify conditions that must (positive) or must not (negative) be satisfied immediately after the current position in the input string. The main difference between positive look-ahead and negative look-ahead is the polarity of the condition.

Positive Look-Ahead:
Syntax: (?=...)
Positive look-ahead asserts that the specified pattern must be present after the current position in the input string.
It matches the current position if the pattern following the look-ahead matches, but it does not consume any characters in the input string.
It is useful when you want to find a match only if it is followed by a specific pattern.
Example: To match a word that is followed by the word "example", you can use the pattern \w+(?= example).

Negative Look-Ahead:
Syntax: (?!...)
Negative look-ahead asserts that the specified pattern must not be present after the current position in the input string.
It matches the current position if the pattern following the look-ahead does not match, but it does not consume any characters in the input string.
It is useful when you want to find a match only if it is not followed by a specific pattern.
Example: To match a word that is not followed by the word "example", you can use the pattern \w+(?! example).

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

Solution - Referring to groups by name rather than by number in a regular expression provides several benefits:

Improved Readability: Using named groups makes the regular expression pattern more readable and self-explanatory. Group names can provide meaningful labels that describe the captured content, making the pattern easier to understand for both the original developer and future maintainers of the code.

Enhanced Code Clarity: Named groups make the code more expressive and self-documenting. By using descriptive names for groups, it becomes clearer what each group represents, enhancing the overall clarity and maintainability of the code.

Flexibility and Robustness: When the structure of the regular expression changes, such as when adding or rearranging groups, using names ensures that the references to those groups remain valid. This improves the robustness of the code as it avoids the need to update the code in multiple places if the group numbers change.

Group Extraction by Name: Referring to groups by name allows easy extraction of specific captured content using the group name. This simplifies post-processing tasks as you can access the captured content directly by its meaningful name, rather than relying on positional indices.

Code Portability: When working with multiple programming languages or regular expression engines, the syntax for referring to groups by number may vary. Using named groups provides a consistent and portable approach, as group names are widely supported across different programming languages and regex implementations.

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

Solution - In a regular expression, named groups can be used to capture and identify repeated items within a target string. However, the capability to identify repeated items using named groups depends on the specific pattern and the context in which it is used.
In the example you provided, "The cow jumped over the moon," if you want to identify repeated words within the string, you can use a backreference with named groups.

```
import re
pattern = r'\b(?P<word>\w+)\b.*\b(?P=word)\b'
text = "The cow jumped over the moon"
matches = re.findall(pattern, text)
repeated_words = set(matches)
print(repeated_words)  # Output: {'the'}
```

In this example, the pattern r'\b(?P<word>\w+)\b.*\b(?P=word)\b' is used to capture a word (\w+) and then match any characters (.*) before finding the same captured word again ((?P=word)). The use of (?P<word>...) defines a named group called "word" to capture the word.

The re.findall() function is used to find all matches of the pattern in the given text. The set of repeated words is then obtained by converting the matches into a set.In this case, the output is {'the'} because the word "the" is repeated in the string.
It's important to note that the specific pattern used and the requirements of identifying repeated items may vary depending on the context and the desired outcome. The pattern provided here demonstrates one approach using named groups to identify repeated words within a string.

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?

Solution - When parsing a string, the Scanner interface in Python's re module provides functionality that the re.findall() feature does not. One important thing that the Scanner interface does for you is it allows you to iterate through the string and match patterns sequentially, providing more control over the parsing process. This is particularly useful when dealing with complex patterns or when you need to perform specific actions based on different matches.

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

Solution – No, a Scanner object in Python does not have to be named "scanner." The choice of variable name is completely up to the programmer and can be any valid variable name according to Python's naming conventions.

While "scanner" is a commonly used variable name for a Scanner object to indicate its purpose, you are free to choose a different name that is more meaningful in the context of your code. It is generally recommended to choose variable names that are descriptive and indicative of the object's purpose to enhance code readability and maintainability.