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?

Ans: Greedy syntax matches the longest possible string, while non-greedy matches the shortest. To transform a greedy pattern to non-greedy, append a "?" after the quantifier. For example, changing "." to ".?" makes it non-greedy.

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?

Ans: Greedy versus non-greedy makes a difference when there are multiple possible matches for a given pattern, and the greedy or non-greedy behavior determines which match is chosen.

If we're looking for a non-greedy match but the only one available is greedy, we can modify the pattern to make it non-greedy by adding a question mark after the quantifier. Alternatively, we can use negative lookaheads or lookbehinds to exclude certain patterns from the match, which can also help to make the match non-greedy.

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?

Ans: 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 unlikely to make any practical difference. Non-capturing groups are typically used to group together parts of a pattern for quantification or alternation purposes, without creating a capture group that can be referenced later.

However, since non-capturing groups do not capture any text, they may be slightly faster than capturing groups in some regex engines. However, the performance gain is likely to be very small and may not be noticeable in most cases.

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

Ans: Non-capturing groups (nontagged categories) are used to group together parts of a regular expression without creating a capture group that can be referenced later. While using a non-capturing group instead of a capturing group is unlikely to have a significant impact on program outcomes, there are some scenarios where using non-capturing groups can be useful:

Ex:

When we are using a complex regex pattern that contains multiple capture groups, and we are using the captured text to perform some post-processing operation. In such cases, using too many capturing groups can make the regex pattern hard to read and debug, and can also slow down the matching process. By using non-capturing groups instead of capturing groups for some parts of the pattern that we don't need to reference later, we can simplify the regex pattern and improve its performance.

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.

Ans: Lookahead assertions (both positive and negative) are used in regular expressions to check if a pattern exists ahead (or behind) a specific position without consuming the characters that the pattern matches. This feature can make a difference in the results of a program in various scenarios, for example:

**Validation of input:** When validating user input, lookahead assertions can be used to check if the input matches a specific pattern without consuming it. For example, if you want to check if a password contains at least one uppercase letter, one lowercase letter, and one number, you can use lookahead assertions to check for the presence of these patterns without consuming them. This allows you to provide feedback to the user if the password does not meet the required criteria without altering the input.

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

Ans: Positive lookahead and negative lookahead are used to check for the existence or non-existence of a pattern ahead of a specific position without consuming any characters. Positive lookahead checks if the pattern exists, while negative lookahead checks if the pattern does not exist.

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

Ans: In a regular expression, it is possible to refer to groups either by number or by name. Referring to groups by name has several benefits over referring to them by number:

a. Clarity and readability: Referring to groups by name makes the regular expression more readable and understandable. It makes the code easier to maintain and update, especially if the regular expression has many groups.

b. Self-documenting: Using named groups in regular expressions is a form of self-documentation. It helps other developers (or even the same developer in the future) understand what the purpose of each group is without having to read the code in detail.

c. Flexibility: Using named groups in regular expressions gives you more flexibility to rearrange, add or remove groups without having to update every reference to that group in the rest of the regular expression. This is particularly useful when dealing with complex regular expressions.

d. Better error handling: When using named groups, the regular expression engine can provide better error messages and debugging information. If there is an error in a regular expression that uses named groups, the error message can indicate which named group caused the problem.

e. Easy access to matches: When using named groups, it is easy to access the matched groups in the result object. The result object will have a property for each named group, making it easy to access the values of each group.

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

Ans: Yes, it is possible to identify repeated items within a target string using named groups in a regular expression. Here is an example regular expression that uses named groups to match repeated words in a string:

\b(?<word>\w+)\b(?:\W+\k<word>\b)+

Let's break down this regular expression:

a. \b(?<word>\w+)\b matches a word boundary, followed by one or more word characters, and then another word boundary. This is captured in a named group called "word".

b. (?:\W+\k<word>\b)+ matches one or more non-word characters, followed by the exact same word that was captured in the "word" group, and then another word boundary. This is repeated one or more times (the plus sign at the end).

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?

Ans: The Scanner interface in Java is used for parsing a string into tokens based on a specified delimiter pattern. One thing that Scanner does is re.findall in Python does not is that it can parse multiple types of data, not just text data.

For example, the Scanner interface can parse data types such as integers, floating-point numbers, and other types that are not just text. It provides methods such as nextInt(), nextDouble(), and nextBoolean() to parse input into their corresponding data types.

On the other hand, re.findall in Python is designed specifically for text data, and it returns a list of all non-overlapping matches of a regular expression pattern in a string. It does not provide a way to parse input into different data types like Scanner does.

Therefore, if you need to parse a string that contains data types other than text, or if you need to parse input into different data types, then Scanner in Java would be a better choice than re.findall in Python.

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

Ans: No, we can use any variable name in python for scanner object,as with any other Python object. However, it is a convention in Python to use lowercase letters and underscores to separate words in variable names.

Here is an example of creating a Scanner object with a variable named my\_scanner:

import scanner

my\_scanner = scanner.Scanner("3 + 4 \* 2")