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?**

A.Greedy syntax grabs the longest match, consuming as much as possible. Non-greedy syntax grabs the shortest match, consuming as little as possible. To change greedy to non-greedy, add a "?" after the quantifier (?, \*, +).

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?**

A. Greedy versus non-greedy matching is primarily relevant in the context of regular expressions, where it determines how much text a pattern should match. Greedy matching attempts to match as much text as possible, while non-greedy (also known as lazy) matching tries to match as little text as possible.

The difference becomes crucial when dealing with patterns that can match multiple parts of a string. For instance, consider the regular expression `.\*end`, which is meant to match any text followed by "end" in a string. In a greedy match, `.\*` will consume as much text as possible, potentially matching more than intended. For example, given the string "start middle end", the greedy match would be "start middle end" instead of just "end".

Non-greedy matching, denoted by adding a `?` after the quantifier (e.g., `.\*?`), resolves this issue. In the same example, a non-greedy match would correctly return "start end".

However, if you specifically require a non-greedy match but the only available option is greedy, you might need to modify your approach. This could involve restructuring your regular expression or using additional logic to achieve the desired outcome. In some cases, it might be necessary to reconsider the design of your pattern or the way you're processing the text to ensure it aligns with your requirements.

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?**

**a.** In a simple match of a string where you're only looking for one match and not doing any replacement, using a non-tagged group versus a tagged group generally doesn't make a practical difference in terms of functionality.

Non-tagged groups are used for grouping purposes but do not capture the matched substring for later use, whereas tagged groups (also called capturing groups) do capture the matched substring for later reference, such as in replacement operations or accessing matched substrings programmatically.

If you're not intending to capture the matched substring for later use, using a non-tagged group is sufficient and might be slightly more efficient since it doesn't incur the overhead of capturing and storing the matched substring. However, the difference in efficiency is likely negligible in most cases unless you're working with very large strings or performing a massive number of string matches.

In terms of the match result, both tagged and non-tagged groups will match the same substrings, so the choice between them mainly depends on whether you need to capture the matched substring for later use**.**

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

A, Imagine a large-scale research project aimed at analyzing social media data to understand public sentiment towards a newly launched product. The project involves collecting vast amounts of text data from various social media platforms such as Twitter, Facebook, and Reddit.

Initially, the research team plans to categorize the data into predefined tags like "positive," "negative," and "neutral" sentiments. However, they also decide to include a nontagged category for any data that doesn't fit neatly into these predefined tags.

As the project progresses, the team notices that a significant portion of the data falls into this nontagged category. Upon closer examination, they realize that this uncategorized data contains nuanced opinions, diverse perspectives, and even emerging trends that were not captured by the predefined tags.

Here's how using the nontagged category could have a significant impact on the program's outcomes:

1. \*\*Richer Insights\*\*: By including the nontagged category, the research team gains access to a wealth of diverse opinions and perspectives that might have been overlooked otherwise. This allows for a more comprehensive understanding of public sentiment towards the product.

2. \*\*Identifying Emerging Trends\*\*: The nontagged category may contain data related to emerging trends or topics that were not initially considered by the research team. By analyzing this uncategorized data, the team can identify and capitalize on these trends, providing valuable insights for product development and marketing strategies.

3. \*\*Fine-tuning Sentiment Analysis Models\*\*: Analyzing the data in the nontagged category allows the research team to identify patterns and trends that can be used to improve sentiment analysis models. By incorporating these insights into their models, they can enhance the accuracy and effectiveness of sentiment analysis for future projects.

4. \*\*Understanding Complex Sentiments\*\*: Some opinions and sentiments expressed in social media data may be complex and nuanced, making them difficult to categorize using predefined tags. The nontagged category provides a space to capture these complex sentiments, enabling a deeper understanding of consumer attitudes and behaviors.

Overall, by incorporating a nontagged category into their analysis, the research team can uncover hidden insights, identify emerging trends, improve sentiment analysis models, and gain a more nuanced understanding of public sentiment towards the newly launched product.

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.**

A. Consider a scenario where you're parsing a text document containing email addresses and trying to extract only the email addresses that end with ".com".

If you use a normal regex pattern without look-ahead, like `\b\w+@\w+\.com\b`, it will consume characters while matching, meaning it will match the entire email address including the ".com" part. However, in a situation where you want to just identify ".com" emails without consuming the entire email address, you could use a look-ahead condition.

For instance, with the pattern `\b\w+@\w+(?=\.com)\b`, the look-ahead `(?=\.com)` asserts that what follows the "@" symbol must be ".com", but it doesn't actually include ".com" in the match. This allows you to just identify the email addresses that end with ".com" without consuming the ".com" part itself.

In practical terms, this means you could use the same pattern to just identify the email addresses with the desired domain without altering the contents of the document or the captured groups in your program.

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

A. Positive lookahead and negative lookahead are both lookahead assertions used in regular expressions, but they serve different purposes:

1. \*\*Positive Lookahead `(x(?=y))`:\*\*

- Matches `x` only if `x` is followed by `y`.

- The `(?=y)` part is the positive lookahead assertion.

- It asserts that whatever follows `x` must be `y`, but it does not include `y` in the match itself.

- For example, the regex `q(?=u)` would match the `q` in "quran" but not in "iq".

2. \*\*Negative Lookahead `(x(?!y))`:\*\*

- Matches `x` only if `x` is not followed by `y`.

- The `(?!y)` part is the negative lookahead assertion.

- It asserts that whatever follows `x` must not be `y`.

- For example, the regex `q(?!u)` would match the `q` in "iq" but not in "quran".

In summary, positive lookahead ensures that a certain pattern is followed by another pattern, while negative lookahead ensures that a certain pattern is not followed by another pattern.

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

A. Referring to groups by name rather than by number in a standard expression can enhance readability and maintainability of the code or expression. Here are some benefits:

1. \*\*Clarity\*\*: Names provide semantic meaning to the groups, making it easier for others (and yourself) to understand the purpose of each group.

2. \*\*Readability\*\*: Descriptive names make the expression more readable, especially for complex patterns or regular expressions where the meaning of numbered groups might not be immediately obvious.

3. \*\*Maintenance\*\*: If the structure of the expression changes, using named groups makes it easier to update the expression without having to adjust the numbering of the groups.

4. \*\*Self-documentation\*\*: Named groups act as self-documentation for the regular expression, providing insight into the purpose of each group without needing additional comments.

5. \*\*Debugging\*\*: When debugging, named groups can provide more informative error messages or logs, indicating which part of the expression failed and why.

Overall, using named groups improves the overall quality of the code or expression by making it more understandable, maintainable, and less error-prone.

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

A. Certainly! You can use regular expressions in Python to identify repeated items within a target string using named groups. Here's how you can do it:

```python

import re

# Define the target string

target\_string = "The cow jumped over the moon moon"

# Define the regular expression pattern with a named group

pattern = r'(?P<repeated\_word>\b\w+\b)\s+(?P=repeated\_word)'

# Find all repeated items within the target string

repeated\_items = re.findall(pattern, target\_string)

# Print the repeated items

print("Repeated items:", repeated\_items)

```

In this code:

- `(?P<repeated\_word>\b\w+\b)` is a named group named `repeated\_word` that matches any word (`\w+`) surrounded by word boundaries (`\b`).

- `\s+` matches one or more whitespace characters.

- `(?P=repeated\_word)` is a backreference to the named group `repeated\_word`, ensuring that the same word is repeated.

When you run this code, it will output:

```

Repeated items: ['moon']

```

This indicates that the word "moon" is repeated within the target 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**?

A. One advantage of using the `Scanner` interface over Python's `re.findall()` function is that `Scanner` provides more flexibility in parsing complex string patterns.

`Scanner` is a class in Java that allows you to parse input strings using regular expressions or custom delimiters. It offers methods like `hasNext()`, `next()`, `next(Pattern pattern)`, etc., which allow you to extract tokens from the input string based on various conditions.

Here are a few things that `Scanner` provides that `re.findall()` does not:

1. \*\*Tokenization\*\*: `Scanner` allows you to tokenize input strings based on specific patterns or delimiters. You can define complex rules for tokenizing strings, which might not be easily achievable with `re.findall()`, especially for non-regular languages.

2. \*\*Multiple Patterns\*\*: With `Scanner`, you can define multiple patterns and scan for each pattern individually. This can be useful when you need to extract different types of information from the same input string.

3. \*\*Custom Parsing Logic\*\*: `Scanner` allows you to implement custom parsing logic using callbacks or custom methods. This gives you more control over how you want to parse and process the input string.

4. \*\*Formatted Input\*\*: `Scanner` supports parsing formatted input, such as numbers in different formats (integers, floating-point numbers, hexadecimal, etc.), dates, times, etc. It provides methods like `nextInt()`, `nextDouble()`, `nextBoolean()`, etc., which simplify parsing formatted input.

Overall, while `re.findall()` is powerful for simple pattern matching and extraction, `Scanner` in Java offers more flexibility and control over parsing complex input strings.

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

A. No, a scanner object does not have to be named "scanner." In programming, especially in languages like Java, "scanner" is often used as a common name for objects that handle input from various sources, such as the keyboard or files. However, you can name your scanner object whatever you want, as long as it follows the naming conventions of the programming language you're using and makes sense within the context of your code.