**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 aims to match as much text as possible, while non-greedy syntax matches as little text as possible.

To transform a greedy pattern into a non-greedy one, the bare minimum effort required is to add a question mark (`?`) after the quantifier (`\*`, `+`, `?`, or `{}`) in the pattern. This question mark changes the behavior from greedy to non-greedy.

For example:

- Greedy pattern: `.\*` (matches as much as possible)

- Non-greedy pattern: `.\*?` (matches as little as possible)

The introduction of the question mark after the quantifier is the minimal change needed to switch from a greedy to a non-greedy pattern.

**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?**

The difference between greedy and non-greedy matching occurs when there are multiple possible matches within a given text, and the pattern allows for flexibility in the matching behavior.

When a pattern is greedy, it tries to match as much text as possible, potentially spanning across multiple occurrences of the pattern. On the other hand, non-greedy matching aims to find the shortest possible match.

If you specifically require a non-greedy match but only a greedy match is available, it means that there are no shorter matches according to the pattern and the given text. In such cases, the non-greedy behavior cannot be achieved, and you would need to modify the pattern or the input text to obtain the desired non-greedy match.

Here's an example to illustrate this scenario:

```python

import re

text = 'Hello, <b>world</b>. This is <b>bold text</b>.'

pattern = r'<b>(.\*?)</b>' # Non-greedy pattern

match = re.search(pattern, text)

if match:

print(match.group(1)) # Output: 'world'

```

In the example above, the text contains two occurrences of `<b>` tags. The pattern `<b>(.\*?)</b>` is a non-greedy pattern that aims to match the content within the first pair of `<b>` tags. In this case, it successfully matches the text `'world'`. If a greedy pattern like `<b>(.\*)</b>` was used instead, it would match the entire string between the first occurrence of `<b>` and the last occurrence of `</b>`, resulting in `'world</b>. This is <b>bold text'` as the match.

If a non-greedy match is not available due to the absence of a shorter match according to the pattern and the given text, you may need to reconsider the pattern or modify the text to achieve the desired non-greedy behavior.

**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 doing any replacement, the use of a non-tagged group (a capturing group without a tag/name) typically does not make any practical difference in terms of the final result.

A non-tagged group is used to group a sequence of patterns together, but it does not assign a specific name or capture the matched substring for later use. If you are not interested in capturing or referencing the matched substring of a group, using a non-tagged group will not have a significant impact on the outcome of the match.

Consider the following example:

```python

import re

text = "Hello, world!"

pattern = r"Hello, (world)!"

match = re.search(pattern, text)

if match:

print(match.group(1)) # Output: "world"

```

In this case, the pattern `Hello, (world)!` contains a capturing group `(world)` which captures the substring "world" within parentheses. The `match.group(1)` retrieves the content captured by the group. Whether the group is tagged or non-tagged, the outcome will be the same.

However, there may be cases where non-tagged groups can still be useful. For example, they can be used for grouping purposes to apply quantifiers, specify alternations, or modify the behavior of a subpattern. Non-tagged groups can also affect the overall structure and readability of the regular expression, making it easier to understand and maintain.

In summary, in a simple match where you are not capturing or referencing the matched substring, the use of a non-tagged group is unlikely to make a practical difference in the final result.

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

Using a non-tagged category, also known as a non-capturing group, can have a significant impact on a program's outcomes in scenarios where you want to group patterns together but do not want to capture or reference the matched substring. Here's an example scenario where non-tagged groups can make a difference:

Scenario: Extracting URL domains without capturing the entire URL

Suppose you have a list of URLs and you want to extract only the domain names from each URL without capturing the entire URL. The URLs can have different formats, including variations in protocols (http, https), subdomains, and top-level domains.

Using a non-tagged group with the appropriate regular expression, you can achieve this task efficiently. Let's consider the following example:

```python

import re

urls = [

"https://www.example.com",

"http://subdomain.example.com",

"https://www.subdomain.example.co.uk",

"http://www.example.org",

]

pattern = r"https?://(?:www\.)?([^.]+)\."

for url in urls:

match = re.search(pattern, url)

if match:

domain = match.group(1)

print(domain)

```

In this example, the regular expression pattern `https?://(?:www\.)?([^.]+)\.` matches the URLs. The non-tagged group `(?:www\.)?` is used to group the optional "www." part of the domain, while the capturing group `([^.]+)` captures the actual domain name.

The use of a non-tagged group `(?:www\.)?` ensures that the "www." part of the domain is not captured and does not appear in `match.group(1)`. Only the domain name itself is captured and printed.

Output:

```

example

subdomain.example

subdomain.example

example

```

In this scenario, using a non-tagged group is crucial to extract only the desired domain names without including the protocol or optional "www." subdomain. The non-tagged group prevents capturing unnecessary parts, resulting in more accurate and desired outcomes.

Overall, using non-tagged groups can be valuable when you need to group patterns together but avoid capturing specific parts, leading to significant differences in program outcomes.

**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 look-ahead condition in regular expressions allows you to specify a pattern that must be followed by another pattern without consuming the characters it examines. This non-consuming behavior of look-ahead assertions can make a difference in program results in scenarios where you want to enforce certain conditions without including them in the final match. Here's an example situation where this behavior can be significant:

Scenario: Validating Password Strength without Including Special Characters

Suppose you need to validate the strength of user passwords based on specific criteria, such as requiring a minimum length and the inclusion of both uppercase and lowercase letters. However, you want to enforce these conditions without including the special characters in the final matched password.

Using look-ahead assertions, you can achieve this requirement efficiently. Consider the following example:

```python

import re

passwords = [

"Abc123!", # Valid password

"password123", # Invalid password (no special character)

"ABCD", # Invalid password (no lowercase letter)

"abcdefg123!" # Invalid password (no uppercase letter)

]

pattern = r"^(?=.\*[a-z])(?=.\*[A-Z])(?=.\*\d)[a-zA-Z\d]{6,}$"

for password in passwords:

match = re.match(pattern, password)

if match:

print(f"{password}: Valid")

else:

print(f"{password}: Invalid")

```

In this example, the regular expression pattern `^(?=.\*[a-z])(?=.\*[A-Z])(?=.\*\d)[a-zA-Z\d]{6,}$` is used to validate the password strength. Let's break down the pattern:

- `^(?=.\*[a-z])`: Positive look-ahead assertion to ensure at least one lowercase letter exists.

- `(?=.\*[A-Z])`: Positive look-ahead assertion to ensure at least one uppercase letter exists.

- `(?=.\*\d)`: Positive look-ahead assertion to ensure at least one digit exists.

- `[a-zA-Z\d]{6,}`: Match the actual password consisting of at least 6 characters, which can be either lowercase, uppercase, or digits.

By using look-ahead assertions, the special characters (e.g., `!` in this case) are not included in the final matched password. However, they are considered as conditions for validating the password strength.

Output:

```

Abc123!: Valid

password123: Invalid

ABCD: Invalid

abcdefg123!: Invalid

```

In this scenario, the non-consuming behavior of the look-ahead assertions allows you to validate the password strength without including the special characters in the final matched password. It ensures the conditions are satisfied without altering the matched password itself.

Thus, by utilizing look-ahead conditions, you can enforce specific criteria without consuming the characters examined, which can significantly impact the results of your program.

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

In regular expressions, both positive look-ahead and negative look-ahead are look-around assertions used to specify conditions that must or must not be followed by a given pattern, without including the pattern in the actual match. The difference lies in the nature of the conditions they enforce.

Positive Look-Ahead (`(?=...)`):

- Syntax: `(?=...)`

- Matches the current position if the specified pattern (look-ahead) matches ahead.

- Does not consume the characters it examines.

- It enforces that the pattern must be followed by the look-ahead pattern.

- Example: `foo(?=bar)` matches "foo" only if it is followed by "bar".

Negative Look-Ahead (`(?!...)`):

- Syntax: `(?!...)`

- Matches the current position if the specified pattern (look-ahead) does not match ahead.

- Does not consume the characters it examines.

- It enforces that the pattern must not be followed by the look-ahead pattern.

- Example: `foo(?!bar)` matches "foo" only if it is not followed by "bar".

To summarize, positive look-ahead asserts that a pattern must be followed by a specified condition, while negative look-ahead asserts that a pattern must not be followed by a specified condition. They both provide a way to enforce additional constraints without including the condition in the actual matched text.

**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: Using named groups in regular expressions enhances the readability and understandability of the pattern. By assigning meaningful names to groups, it becomes easier to comprehend the purpose and intent of each captured group.

2. Code Maintenance: Named groups make the regular expression pattern more maintainable. If the pattern needs to be modified or extended in the future, using named groups allows you to refer to captured groups by their descriptive names, making it clear which parts of the pattern are affected.

3. Self-Documenting Patterns: Named groups serve as documentation within the regular expression itself. By using descriptive names, the pattern becomes self-explanatory, reducing the need for additional comments or external documentation.

4. Flexibility in Rearranging Patterns: When using named groups, the order of the groups in the regular expression pattern becomes less critical. You can rearrange the groups within the pattern without affecting the code that references the captured groups by name.

5. Avoiding Dependency on Group Numbering: Referring to groups by name eliminates the reliance on group numbering. Group numbers can change if the pattern is modified, which can introduce bugs and errors if the code is not updated accordingly. Named groups provide a more robust and stable approach, unaffected by group reordering or additions.

6. Accessibility in Match Objects: When using named groups, the match object returned by regular expression functions provides access to captured groups through their names, allowing easier retrieval and manipulation of specific captured content.

To summarize, referring to groups by name in regular expressions enhances readability, maintainability, and self-documentation of the pattern. It provides flexibility, reduces dependencies on group numbering, and allows for easier access and manipulation of captured content through named 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 identify repeated items within a target string using named groups in regular expressions. However, the example you provided, "The cow jumped over the moon," does not contain any repeated items.

To demonstrate how named groups can be used to identify repeated items, let's consider a different example. Let's say we have the following string: "She sells seashells by the seashore."

If we want to identify any repeated words in this string, we can use named groups in a regular expression. Here's an example in Python:

```python

import re

string = "She sells seashells by the seashore."

pattern = r"\b(?P<word>\w+)\b.\*\b(?P=word)\b"

matches = re.findall(pattern, string)

for match in matches:

print("Repeated word:", match)

```

In this example, the regular expression pattern `\b(?P<word>\w+)\b.\*\b(?P=word)\b` is used. It defines a named group called "word" (`(?P<word>\w+)`) to match any word character, and `\b` is used to match word boundaries. The `.\*` in the middle allows for any characters between the repeated words. Finally, `(?P=word)` is a backreference to the named group, ensuring that the same word is repeated.

When running the above code, it will output:

```

Repeated word: seashells

Repeated word: the

```

This shows that the words "seashells" and "the" are repeated in the given 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?**

When parsing a string, the `Scanner` interface in Java provides several advantages over the `re.findall` feature in Python's regular expressions:

1. Tokenization: The `Scanner` interface in Java allows you to easily tokenize a string by breaking it into individual elements or tokens based on specific delimiters. It provides methods like `next()`, `nextLine()`, and `useDelimiter()` that simplify the process of extracting tokens from the input. In contrast, `re.findall` in Python is primarily used for pattern matching and does not provide built-in tokenization capabilities.

2. Typed Input Retrieval: The `Scanner` interface in Java provides methods like `nextInt()`, `nextDouble()`, and `nextBoolean()` to directly retrieve typed input from the string tokens. These methods automatically convert the input to the desired data type, simplifying the parsing process. On the other hand, `re.findall` in Python returns matched patterns as strings, requiring additional manual conversion or parsing if different data types are needed.

3. Sequential Parsing: With the `Scanner` interface, you can parse a string sequentially, moving through the input one token at a time. This allows for more granular control over the parsing process, enabling you to handle different types of input and perform specific actions based on the token being processed. In contrast, `re.findall` returns all matches at once, making it less suitable for sequential parsing scenarios.

4. Error Handling: The `Scanner` interface provides methods like `hasNext()` and `hasNextInt()` to check for input validity before retrieval. These methods allow you to handle potential parsing errors or unexpected input gracefully by providing appropriate error messages or fallback mechanisms. On the other hand, `re.findall` does not have built-in error handling mechanisms and assumes that the provided pattern will match successfully.

While `re.findall` is a powerful tool for pattern matching and extracting multiple occurrences of a pattern from a string, the `Scanner` interface in Java provides a more comprehensive set of features for parsing and tokenizing input strings, including typed input retrieval, sequential parsing, and error handling.

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

No, a `Scanner` object does not have to be named "scanner." You can choose any valid identifier as the name for your `Scanner` object.

In Java, when you create an instance of the `Scanner` class, you can assign it to any variable name you prefer, as long as it follows the rules for naming variables in Java. The variable name you choose should be meaningful and descriptive, making it easier to understand the purpose of the `Scanner` object in your code.

Here's an example of creating a `Scanner` object with a different variable name:

```java

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Scanner inputScanner = new Scanner(System.in);

// Rest of the code...

}

}

```

In the above example, the `Scanner` object is named `inputScanner` instead of just `scanner`. You can use this object, referred to by the variable name `inputScanner`, to interact with the input stream and perform various parsing operations.

Remember that the choice of variable name is up to you, but it is generally recommended to use a name that accurately reflects the purpose or context of the `Scanner` object in your code to enhance code readability and maintainability.