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 vs. Non-Greedy Syntax:

- Greedy: Matches as much as possible (expands to the right).

- Non-Greedy: Matches as little as possible (expands minimally to the right).

Transformation from Greedy to Non-Greedy:

Add a `?` after the quantifier or repetition character.

For example, change `\*` to `\*?` or `+` to `+?`.

This slight addition of the `?` makes the quantifier non-greedy, causing it to match the smallest possible substring that satisfies the pattern, rather than the largest possible substring that satisfies the 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?

ANS: Greedy versus non-greedy matching makes a difference when there are multiple occurrences of a pattern in the input text that satisfy the regular expression. The distinction lies in how the regex engine matches and expands the pattern to find a valid match.

Greedy Matching:

- In greedy matching, the regex engine tries to find the longest possible match that satisfies the pattern.

- It starts by matching as many characters as possible and then backtracks if necessary to satisfy the rest of the pattern.

Non-Greedy (Lazy) Matching:

- In non-greedy matching, the regex engine tries to find the shortest possible match that satisfies the pattern.

- It starts by matching as few characters as possible and expands the match only if needed to satisfy the rest of the pattern.

Example (Greedy vs. Non-Greedy):

Suppose we have the input text: "abcbcbc"

A. Greedy: Pattern is `a.\*c`.

- Greedy matching would find the longest possible match: "abcbcbc".

- It matches from the first "a" to the last "c" in the text.

B. Non-Greedy: Pattern is `a.\*?c`.

- Non-greedy matching would find the shortest possible match: "abc".

- It matches from the first "a" to the first "c" in the text.

If you are looking for a non-greedy match but only a greedy match is available in the input text, the regex engine will still perform a match. In this case, the greedy match will be returned, and it will be the longest possible match that satisfies the pattern.

To ensure a non-greedy match when only a greedy option is available, you need to modify the quantifier by adding a `?` after it. This will explicitly instruct the regex engine to perform non-greedy matching. By doing so, you can change a greedy pattern into a non-greedy one without needing to alter the rest of the regular expression.

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, where you are only looking for one match and not using any captured groups for extraction or replacement, the use of a non-capturing group (a non-tagged group) is unlikely to make any practical difference.

A non-capturing group is denoted by `(?:pattern)`, where `pattern` represents the subpattern you want to group. The purpose of a non-capturing group is primarily to group subpatterns together for applying quantifiers or alternations without capturing the matched substring as a separate group.

In the context of simple matching without capturing groups, both a capturing group and a non-capturing group will have the same effect on the actual match result. That is, they will not affect the overall match, and the matched substring will be the same regardless of whether you use a capturing group or a non-capturing group.

Example:

import re

# Simple match for "apple"

pattern\_capturing = r'(apple)'

pattern\_non\_capturing = r'(?:apple)'

text = "I have an apple and an orange."

# Matching with capturing group

match\_capturing = re.search(pattern\_capturing, text)

print("Capturing group:", match\_capturing.group()) # Output: Capturing group: apple

# Matching with non-capturing group

match\_non\_capturing = re.search(pattern\_non\_capturing, text)

print("Non-capturing group:", match\_non\_capturing.group()) # Output: Non-capturing group: apple

As you can see in the example above, both patterns `(apple)` and `(?:apple)` result in the same matched substring "apple" when used with `re.search()`. In this case, since you are not interested in capturing the matched substring for further use, the choice between a capturing group and a non-capturing group won't have any practical impact on the outcome.

However, if you plan to use the captured groups later for extraction or replacement, then using a capturing group becomes important. Otherwise, if the sole purpose is to group subpatterns without capturing, you can use a non-capturing group for clarity and to avoid unnecessary storage of captured groups.

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

ANS: A scenario where using a non-capturing group (a non-tagged category) would have a significant impact on the program's outcomes is when the regular expression involves complex alternations or quantifiers with capturing groups.

Consider the following scenario where we want to extract URLs from a given text. URLs can have different formats, and we want to capture both HTTP and HTTPS URLs. We can use an alternation with capturing groups to extract the URLs and their corresponding schemes:

import re

text = "Visit our website at http://www.example.com and also check out https://www.example.com."

# Using capturing groups in the regular expression

pattern\_with\_capturing = r'(http|https)://(www\.)?example\.com'

matches\_with\_capturing = re.findall(pattern\_with\_capturing, text)

print("Matches with capturing:", matches\_with\_capturing)

# Output: [('http', 'www.'), ('https', 'www.')]

In this case, the capturing groups capture the scheme (HTTP/HTTPS) and the presence of "www." for each URL.

Now, let's consider the same scenario using non-capturing groups instead:

text = "Visit our website at http://www.example.com and also check out https://www.example.com."

# Using non-capturing groups in the regular expression

pattern\_with\_non\_capturing = r'(?:http|https)://(?:www\.)?example\.com'

matches\_with\_non\_capturing = re.findall(pattern\_with\_non\_capturing, text)

print("Matches with non-capturing:", matches\_with\_non\_capturing)

# Output: ['http://www.example.com', 'https://www.example.com']

In this case, the non-capturing groups allow us to ignore the capturing of the individual components (scheme and "www." part) and capture the complete URLs only. This simplifies the output, as we are now only interested in the full URLs.

The difference in outcomes between the two scenarios becomes significant when you need to process the captured data further. Using non-capturing groups in this case allows you to get a cleaner and more concise result, focusing only on the complete URLs without capturing the individual components separately. While this example might seem relatively simple, in more complex scenarios with multiple capturing groups, using non-capturing groups can lead to more efficient and readable code.

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: A look-ahead condition in a regular expression is a non-consuming assertion that checks if a particular subpattern exists ahead of the current position, without actually including it in the match. This feature allows you to create more complex and specific patterns without affecting the overall match.One situation where look-ahead conditions can make a significant difference in the results of your program is when you want to match a pattern that satisfies specific conditions but exclude certain substrings from the final match.

Let's consider a scenario where we have a list of email addresses, and we want to find all email addresses that are not from certain domains (e.g., exclude addresses from gmail.com and yahoo.com).

Using a look-ahead condition, we can define a regular expression that looks for email addresses but excludes those with specific domains:

import re

# Sample list of email addresses

email\_list = [

"john.doe@example.com",

"jane\_smith@gmail.com",

"mark.smith@yahoo.com",

"test@example.com"

]

# Regular expression with a negative look-ahead to exclude gmail.com and yahoo.com

pattern = r'\b[A-Za-z0-9.\_%+-]+@(?!gmail\.com|yahoo\.com)[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b'

# Find all matches that satisfy the pattern

matches = [match.group() for email in email\_list if (match := re.search(pattern, email))]

print(matches)

# Output: ['john.doe@example.com', 'test@example.com']

In this example, the regular expression `r'\b[A-Za-z0-9.\_%+-]+@(?!gmail\.com|yahoo\.com)[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b'` uses a negative look-ahead `(?!gmail\.com|yahoo\.com)` to assert that the email address should not be followed by "gmail.com" or "yahoo.com." The look-ahead condition allows the regex to check for this condition without actually consuming the characters representing those domains. As a result, email addresses from other domains are included in the final match.

Without using a look-ahead, you might need to resort to more complex capturing groups and additional post-processing steps to filter out unwanted domains from the final results.Using look-ahead conditions can make your regex patterns more concise, efficient, and expressive, especially in situations where you want to include or exclude specific substrings based on certain conditions without affecting the overall match.

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

ANS: In regular expressions, both positive look-ahead and negative look-ahead are types of look-ahead assertions that allow you to specify conditions that must (positive look-ahead) or must not (negative look-ahead) be present ahead of the current position in the input text, without actually including them in the match.

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

- Syntax: `(?=pattern)`

- A positive look-ahead assertion is used to assert that a specific pattern must be present ahead of the current position in the input text for the match to occur.

- It is a non-consuming assertion, meaning it does not consume characters in the input text while checking the condition.

- The match continues only if the pattern specified inside the look-ahead assertion is found, but the pattern itself is not included in the final match.

- Positive look-ahead allows you to ensure that certain conditions are met without affecting the overall match.

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

- Syntax: `(?!pattern)`

- A negative look-ahead assertion is used to assert that a specific pattern must not be present ahead of the current position in the input text for the match to occur.

- Like positive look-ahead, it is a non-consuming assertion and does not consume characters while checking the condition.

- The match continues only if the pattern specified inside the negative look-ahead assertion is not found.

- Negative look-ahead allows you to exclude certain patterns from the match.

Example of Positive Look-Ahead:

import re

text = "hello123world"

pattern = r'\d+(?=[a-z]+)' # Matches digits that are followed by lowercase letters

matches = re.findall(pattern, text)

print(matches) # Output: ['123']

Example of Negative Look-Ahead:

import re

text = "apple orange banana"

pattern = r'\b\w+(?!\sorange\b)' # Matches words that are not followed by 'orange'

matches = re.findall(pattern, text)

print(matches) # Output: ['apple', 'banana']

In the positive look-ahead example, the pattern `\d+(?=[a-z]+)` matches one or more digits that are followed by one or more lowercase letters. The positive look-ahead `(?=[a-z]+)` ensures that the digits are indeed followed by lowercase letters but does not include the lowercase letters in the match.

In the negative look-ahead example, the pattern `\b\w+(?!\sorange\b)` matches words that are not followed by the word "orange." The negative look-ahead `(?!\\sorange\b)` ensures that the word is not followed by "orange" but does not include "orange" in the match.

Both positive and negative look-ahead assertions are powerful tools for creating more complex and specific regular expressions, allowing you to impose conditions on your matches without affecting the content actually included in the final match.

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

ANS: Referring to groups by name rather than by number in a standard expression provides several benefits that can greatly improve the readability, maintainability, and clarity of your regular expressions and code:

I. \*\*Readability\*\*: Group names make the regex pattern more self-explanatory. Instead of referring to groups by their numeric index (e.g., `\1`, `\2`, etc.), using descriptive names allows you and other developers to easily understand the purpose and intent of each group.

II. \*\*Maintainability\*\*: When you modify the regex pattern, especially if you add or remove groups, using group names ensures that you don't need to update the indices of referenced groups throughout your code. This reduces the risk of introducing errors when modifying the regex.

III. \*\*Self-Documenting Code\*\*: Group names act as inline documentation within the regular expression, making it clear what each group represents. This improves code readability and makes the purpose of the regular expression more apparent to others who might read or maintain the code.

IV. \*\*Flexibility\*\*: Using group names allows you to reorder or add new groups in the regex pattern without affecting the referencing code. This flexibility can be helpful when evolving and expanding your regex over time.

V. \*\*Referring to Multiple Groups\*\*: When your regular expression contains multiple groups of the same type (e.g., capturing multiple named entities), using group names makes it easier to distinguish between them when referencing or processing the matches.

VI. \*\*Named Capture Access\*\*: Some regex libraries or functions support named capture access, where you can access matched groups directly by their names. This makes post-processing of the matches more intuitive and readable.

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

ANS: Yes, you can identify repeated items within a target string using named groups in a regular expression. Named groups allow you to capture and name specific parts of a pattern, and you can use backreferences to refer to those named groups later in the regex pattern.

To identify repeated items within a target string, you can use a named backreference to match the same text that was previously captured by a named group. This technique allows you to find and match repeated occurrences of the same text within the string.

Here's an example of how you can use named groups and backreferences to identify repeated items within the target string "The cow jumped over the moon":

import re

# Pattern with named group and named backreference

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

# Target string

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

# Find all repeated words using findall()

matches = re.findall(pattern, target\_string)

# Print the repeated words

print(matches) # Output: ['the']

In this example, the regular expression pattern `r'(?P<word>\b\w+\b).\*\b(?P=word)\b'` contains a named group `(?P<word>\b\w+\b)` that captures a word boundary followed by one or more word characters. The `.\*` in the middle matches any characters in between the repeated words. The named backreference `(?P=word)` later in the pattern ensures that the text matched by the named group `word` is repeated in the string.

When using `re.findall()`, the function returns a list of all occurrences of the repeated word within the target string. In this example, it finds and returns the repeated word "the."

Using named groups and backreferences allows you to perform more advanced pattern matching and extraction tasks, including identifying repeated items, without explicitly knowing the repeated items beforehand. It makes your regex pattern more dynamic and adaptable to different input strings.

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 Python provides a more advanced and flexible way of parsing strings compared to the `re.findall()` function. One thing that the `Scanner` interface does for you, which `re.findall()` does not, is that it allows you to define more complex patterns and perform custom processing for each matched token.

The `Scanner` interface is part of the `re` module in Python and is used for creating customized lexers or tokenizers. It works by defining a sequence of regular expressions (patterns) and corresponding callback functions that are executed when a pattern is matched in the input string. This gives you fine-grained control over how the string is parsed and allows you to handle each token differently based on its type.

The `re.findall()` function, on the other hand, is primarily used for simple text extraction tasks. It returns all occurrences of a given pattern in the input string as a list, but it does not provide any facility for custom processing or handling different types of tokens separately.

Here's an example to illustrate the difference:

Using `re.findall()` for simple text extraction:

import re

text = "The quick brown fox jumps over the lazy dog."

# Using re.findall() to extract all words

words = re.findall(r'\b\w+\b', text)

print(words) # Output: ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']

Using `Scanner` for custom tokenization:

import re

class Tokenizer:

def \_\_init\_\_(self):

self.scanner = re.Scanner([

(r'\b\w+\b', self.handle\_word), # Match words and call handle\_word

(r'\s+', None), # Ignore whitespace

(r'.', self.handle\_other) # Handle other characters

])

def tokenize(self, text):

return self.scanner.scan(text)[0]

def handle\_word(self, scanner, token):

print("Word:", token)

def handle\_other(self, scanner, token):

print("Other:", token)

text = "The quick brown fox jumps over the lazy dog."

# Using custom Tokenizer to tokenize the input text

tokenizer = Tokenizer()

tokenizer.tokenize(text)

Output using the `Scanner`:

Word: The

Word: quick

Word: brown

Word: fox

Word: jumps

Word: over

Word: the

Word: lazy

Word: dog

As you can see, with the `Scanner` interface, you can define specific handling functions for different patterns (e.g., words, whitespace, other characters), allowing you to customize the parsing process according to the token type. This level of control and customization is not possible with `re.findall()`, which only extracts and returns all occurrences of a pattern without any post-processing or custom handling.

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

ANS: No, a `Scanner` object does not have to be named "scanner." In Python, like any other object, you can choose any valid variable name to refer to a `Scanner` object or any other instance of a class. The name "scanner" used in the examples I provided earlier was simply a descriptive name for the object, but you are free to choose a different name that suits your code and makes it more readable.

When working with the `Scanner` interface or any other class, it's a good practice to choose variable names that are meaningful and descriptive of the object's purpose or functionality. This helps improve code readability and makes it easier for other developers (including your future self) to understand the code.

For instance, if you are building a lexer or tokenizer for processing mathematical expressions, you might name your `Scanner` object as "math\_lexer" or "expression\_tokenizer" to indicate its role in the program.

Here's an example of how you can create and use a `Scanner` object with a different variable name:

import re

class Tokenizer:

def \_\_init\_\_(self):

self.lexer = re.Scanner([

(r'\b\w+\b', self.handle\_word), # Match words and call handle\_word

(r'\s+', None), # Ignore whitespace

(r'.', self.handle\_other) # Handle other characters

])

def tokenize(self, text):

return self.lexer.scan(text)[0]

def handle\_word(self, scanner, token):

print("Word:", token)

def handle\_other(self, scanner, token):

print("Other:", token)

text = "The quick brown fox jumps over the lazy dog."

# Using a custom Tokenizer with a different variable name

expression\_tokenizer = Tokenizer()

expression\_tokenizer.tokenize(text)

The output will be the same as before, but now the `Scanner` object is named "lexer" instead of "scanner." Remember, the choice of variable names is flexible as long as it adheres to Python's naming rules and makes your code more understandable to yourself and others.