**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 matches as much as possible, while non-greedy (or lazy) syntax matches as little as possible.

To transform a greedy pattern to a non-greedy one, add a question mark "?" after the quantifier (such as "\*" or "+").

For example, to change a greedy pattern like .\* to a non-greedy one, you can add a question mark to get .\*?. This tells the regex engine to match as little as possible instead of as much as possible.

In Python, you can use these syntaxes in regular expressions, such as re.findall(pattern, string).

**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?**

Greedy versus non-greedy matching makes a difference when the pattern contains multiple matches and the quantifier (such as "\*" or "+") can match more than one.

For example, given the string "hello world" and the pattern "h.\*?d", the non-greedy match will return "helld" while the greedy match will return "hello world".

If you're looking for a non-greedy match but the only one available is greedy, you can use a negative character set to exclude the characters you don't want to match. For example, if you want to match the first occurrence of "world" in "hello world", you can use the pattern "w[^o]\*d" to match "world" while excluding the "o" in "hello".

In Python, you can use regular expressions with greedy or non-greedy syntax using the re module. You can also use negated character sets to exclude certain characters from a match.

**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 that looks for only one match and does not do any replacement, using a non-capturing group (a group that is not captured and does not create a numbered reference) is unlikely to make any practical difference.

Non-capturing groups are typically used for grouping and alternation purposes in regular expressions, where the group is not intended to be captured or referred to later in the expression.

However, in a simple match where there is only one match and no other capturing groups are used, the use of a non-capturing group is equivalent to using a capturing group, and there would be no practical difference between the two.

In Python, you can use capturing groups by enclosing the group with parentheses, and non-capturing groups by using the syntax (?:pattern). Both types of groups can be used in simple matches, but non-capturing groups are typically used for more complex regular expressions.

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

Using a non-capturing group (a nontagged category) can have a significant impact on the program's outcomes when the regular expression is used in a larger pattern that contains other capturing groups.

In such a scenario, using a non-capturing group can avoid interfering with the numbering of the capturing groups and can make the regular expression more readable and maintainable.

For example, suppose you have a regular expression pattern that matches an email address and captures the username and domain name separately:

pattern = r'(\w+)@(\w+\.\w+)'

If you later need to add an optional flag to the domain name (such as ".com" or ".edu"), you might be tempted to modify the pattern like this:

pattern = r'(\w+)@(\w+(\.\w+)?\.com|\w+(\.\w+)?\.edu)'

However, this modification would cause the numbering of the capturing groups to change, which could break any code that refers to the captured groups by number.

To avoid this problem, you could use a non-capturing group to group the optional flag separately:

pattern = r'(\w+)@(\w+(?:\.\w+)?\.(?:com|edu))'

This pattern uses non-capturing groups to group the optional flag and the alternatives, and does not affect the numbering of the capturing groups. This can make the regular expression more maintainable and less error-prone in the long run.

In Python, you can use non-capturing groups by using the syntax (?:pattern) in a regular expression.

**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 assertion in a regular expression is used to match a pattern only if it is followed by another pattern, without consuming the characters that match the second pattern.

One situation where this could make a difference in the results of a program is when you want to match a pattern that is preceded or followed by another pattern, but you don't want to include the preceding or following pattern in the match.

For example, suppose you have a string that contains a list of email addresses separated by commas and enclosed in parentheses:

text = "(john@example.com, alice@example.com, bob@example.com)"

If you want to extract the email addresses from the string without including the enclosing parentheses or the commas, you could use a look-ahead assertion to match the "@" character only if it is followed by a non-whitespace character:

import re

pattern = r'\b\w+@\w+\.\w+(?=\S)'

emails = re.findall(pattern, text)

print(emails)

This regular expression matches an email address that is preceded by a word boundary (\b) and followed by a look-ahead assertion that checks for a non-whitespace character. This allows the pattern to match only the email address and not the preceding comma or the following whitespace.

In this case, using a look-ahead assertion allows you to extract the email addresses from the string without including the preceding or following characters in the match, which could be important if you need to process the email addresses further in your program.

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

In regular expressions, a look-ahead assertion is used to match a pattern only if it is followed by another pattern, without consuming the characters that match the second pattern.

Positive look-ahead and negative look-ahead are two types of look-ahead assertions that differ in the condition they check.

Positive look-ahead assertion ((?=pattern)) matches a pattern only if it is followed by another pattern. It does not include the second pattern in the match.

For example, the regular expression foo(?=bar) would match "foo" only if it is followed by "bar", but would not include "bar" in the match.

Negative look-ahead assertion ((?!pattern)) matches a pattern only if it is not followed by another pattern. It does not include the second pattern in the match.

For example, the regular expression foo(?!bar) would match "foo" only if it is not followed by "bar".

In Python, you can use look-ahead assertions in regular expressions by using the syntax (?=pattern) for positive look-ahead and (?!pattern) for negative look-ahead.

For example, to match all words that are followed by a comma but not followed by the word "and", you could use the regular expression (?=\w+,)(?!\sand\b)\w+. This regular expression matches a word that is followed by a comma using positive look-ahead, but only if it is not followed by the word "and" using negative look-ahead.

**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 has several benefits:

1. Clarity: Using named groups makes your regular expressions more readable and easier to understand. By giving a meaningful name to each group, you can convey the purpose of the group more clearly than with a number.
2. Flexibility: If the regular expression changes and the group numbers are re-ordered or new groups are added, referring to groups by name ensures that your code will still work correctly, whereas referring to groups by number can easily break.
3. Maintainability: If you have a long and complex regular expression with many groups, it can be difficult to remember which group number corresponds to which part of the expression. Using named groups makes it easier to maintain your code and to make changes to the regular expression without introducing bugs.
4. Reusability: If you have multiple regular expressions that use the same groups, using named groups makes it easy to reuse the groups across different expressions.

In Python, you can use named groups in a regular expression by using the syntax (?P<name>pattern) to name a group. You can then refer to the named group in your code by using the syntax match.group('name') or match.groupdict()['name'] to retrieve the matched text for the named group.

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

Yes, you can use named groups to identify repeated items within a target string. One way to do this is by using back-references to match the same text that was previously matched by a named group.

For example, suppose you have the string "The cow jumped over the moon" and you want to find all words that are repeated in the string. You can use a regular expression with a named group to match each word in the string, and then use a back-reference to match any subsequent occurrences of the same word:

import re

text = "The cow jumped over the moon"

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

matches = re.findall(pattern, text)

print(matches)

In this regular expression, \b(?P<word>\w+)\b matches a word and captures it in a named group called "word". .\* matches any characters between the two occurrences of the word, and \b(?P=word)\b matches any subsequent occurrence of the same word.

The back-reference (?P=word) matches the same text that was previously matched by the named group "word". By using a named group and a back-reference, you can easily identify repeated items within the target string.

When you run this code, it will output a list containing the repeated words in the string

['the']

In this example, "the" is the only word that is repeated in the 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?**

The Scanner interface and the re.findall function are both tools for parsing strings in Python, but they work in different ways and offer different capabilities.

One thing that the Scanner interface can do for you that re.findall cannot is to perform more complex parsing of the input string. The Scanner interface provides a way to define a set of regular expressions that can match different patterns in the input string and perform different actions based on the match.

For example, suppose you have a string that contains a list of words separated by commas, and you want to parse the string and create a list of the words. You could use re.findall to find all occurrences of words in the string, but you would still need to manually split the string and remove any extraneous characters like commas.

Using the Scanner interface, you can define a regular expression that matches individual words and use it to scan the input string, returning each word one at a time. Here's an example:

import re

input\_string = "foo, bar, baz"

scanner = re.Scanner([

(r'\w+', lambda scanner, token: token),

(r'\s\*,\s\*', lambda scanner, token: None),

])

words, \_ = scanner.scan(input\_string)

print(words)

In this example, the Scanner is defined with two regular expressions: r'\w+' matches any sequence of word characters, and r'\s\*,\s\*' matches any sequence of whitespace characters followed by a comma followed by any sequence of whitespace characters. The lambda functions are used to handle the matches: the first lambda function simply returns the matched token, and the second lambda function ignores the match.

When you run this code, it will output a list containing the words in the input string:

['foo', 'bar', 'baz']

The Scanner interface provides a way to perform more complex parsing of input strings by defining a set of regular expressions that can match different patterns and perform different actions based on the matches. This makes it a more powerful tool for parsing strings than re.findall, which simply returns all matches of a given regular expression.

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

No, a Scanner object in Python does not have to be named scanner. The name of the object can be anything you choose, as long as it follows the rules for naming variables in Python.

When you create a Scanner object, you can assign it to a variable with any name you want, like this:

import re

my\_scanner = re.Scanner([

(r'\d+', lambda scanner, token: int(token)),

(r'[a-zA-Z]+', lambda scanner, token: token.lower()),

])

In this example, we create a Scanner object and assign it to a variable called my\_scanner. The Scanner object is defined with two regular expressions that match numbers and letters, respectively, and the lambda functions convert the matched tokens to integers (in the case of numbers) and lowercase strings (in the case of letters).

You can use the my\_scanner variable to scan input strings and perform the defined actions on the matched tokens, like this:

input\_string = '123 foo BAR 456'

tokens, \_ = my\_scanner.scan(input\_string)

print(tokens)

This code will output:

[123, 'foo', 'bar', 456]

So, to summarize, you can name a Scanner object in Python anything you like, as long as the name follows the rules for naming variables in Python.