**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 and non-greedy syntax refer to the behavior of a regular expression (regex) when trying to match a pattern in a string of text. A greedy regex will match as much text as possible, while a non-greedy regex will match as little text as possible.

For example, consider the regex pattern .\* (a dot followed by an asterisk), which matches any character (the dot) zero or more times (the asterisk). In a greedy regex, this pattern would match the entire string of text. In a non-greedy regex, it would only match the smallest possible string that satisfies the pattern.

To convert a greedy regex to a non-greedy regex in Python, you can add the ? character after the quantifier (the asterisk in this example). So the non-greedy version of the above pattern would be .\*?. This tells the regex engine to match as few characters as possible instead of as many as possible.

In general, to convert a greedy regex to a non-greedy regex, you just need to add the ? character after the quantifier. This works for any quantifier, not just the asterisk. For example, +? would make the + quantifier non-greedy, ?? would make the ? quantifier non-greedy, and so on.

**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 and non-greedy regexes can produce different matches depending on the input string. For example, consider the following regex pattern: .\*foo. In a greedy regex, this pattern would match the entire input string barfoobar, since it satisfies the pattern of any character followed by "foo". In a non-greedy regex, however, it would only match the substring barfoo, since that is the smallest possible string that satisfies the pattern.
* When using a regex pattern to search for a match in a string, it is important to consider whether a greedy or non-greedy match is desired. If a greedy match is desired, then no further action is necessary, as most regex engines are greedy by default. However, if a non-greedy match is desired, then the ? character must be added after the relevant quantifier in the regex pattern.
* In the case where the only available match is greedy, then the only way to obtain a non-greedy match would be to modify the input string so that it no longer satisfies the pattern in a greedy manner.

**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 general, non-tagged groups are useful when you want to group together part of a regex pattern for the purpose of applying a quantifier or other regex operator, but you do not want to capture the matched text. This can be useful in scenarios where you want to perform a replacement operation, since the captured text can be accessed and used in the replacement string. In a simple match scenario where no replacements are performed, however, non-tagged groups are not likely to provide any practical benefits.

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

A non-tagged group (also known as a non-capturing group) is useful when you want to group together part of a regex pattern for the purpose of applying a quantifier or other regex operator, but you do not want to capture the matched text. In this case, the group serves as a way to organize the regex pattern and make it easier to read and understand, but it does not affect the outcome of the match.

One scenario in which using a non-tagged group could have a significant impact on the program's outcomes is when you are using a regex pattern to perform a search-and-replace operation on a string of text. In this case, the captured text from a capturing group can be accessed and used in the replacement string, which allows you to perform more complex and sophisticated replacement

**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.**

One potential situation in which using a look-ahead condition in a regular expression could make a difference in the results of a Python program is when you want to match a pattern only if it is followed by a specific sequence of characters, but you don't want to include the characters in the matched pattern.

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

In regular expressions, a positive look-ahead asserts that the pattern immediately following the look-ahead should match. A negative look-ahead asserts that the pattern immediately following the look-ahead should not match.

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

Referring to groups by name in a standard expression in python can make your code more readable and easier to understand. It also allows you to refer to the group multiple times in the same expression, which can be useful if you need to use the group multiple times or if you want to use a more descriptive name for the group. Additionally, using named groups can make it easier to modify your code in the future, as you can simply change the name of the group in the expression rather than having to update the group's position or number. Overall, using named groups can improve the maintainability and readability of your code.

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

Yes, it is possible to identify repeated items within a target string using named groups in a regular expression. For example, you could use the following regular expression to identify the repeated word "the" in the target string "The cow jumped over the moon":

(?P<the>\bthe\b)(?P=the)

This regular expression uses a named group (?P<the>) to match the word "the" in the target string. It then uses a backreference (?P=the) to match the same word again. This regular expression would only match the second occurrence of the word "the" in 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?**

The Scanner interface provides a more powerful and convenient way to parse strings compared to the re.findall method. It allows you to extract multiple pieces of information from a single input string and specify the data types of the extracted values, which can make it easier to work with the extracted data.

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

No, a scanner object does not have to be named "scanner". In Python, you can give an object any name you want, as long as the name is a valid Python identifier and is not already being used for another object. For example, you could name your scanner object "my\_scanner" or "data\_scanner" instead of "scanner". The choice of name is up to you and should reflect the purpose or usage of the object in your code.