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: Greedy matching tries to match as much as possible. It means that the pattern will match the
    longest possible substring that satisfies the given expression.
    Greedy behavior is indicated by using * or + after a character
Non-greedy: Non-greedy matching, also known as lazy or reluctant matching, tries to match as little as possible

To transform a greedy pattern into a non-greedy one, the bare minimum effort required is to introduce or change the ?
character immediately 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?

```
Greedy matching tries to match as much as possible, while non-greedy matching tries to match as little as possible.
```

Q3. In a simple match of a string, which looks only for one match and does not do any replacement, is the use of nontagged group likely to make any practical difference?

```
 In a simple match of a string, which looks only for one match and does not do any replacement,
 the use of nontagged group do not make any practical difference
```

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

```
IN a text document containing various email addresses, and you want to extract the username and
```

```
domain name from each email address. The email addresses are in the format
"username@domain.com".

Using regular expressions, you can define a pattern with non-tagged capturing
groups to extract the username and
domain name separately:

Pattern: (\w+)@(\w+\.\w+)

In this pattern, the first non-tagged group (\w+) captures the username, and the
second non-tagged group (\w+\.\w+)
captures the domain name
```

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.

```
if we have a list of email addresses and you want to match only the email
addresses that are followed by
the domain "example.com", but you don't want to include the domain itself in the
match.

Using a look-ahead condition in the regex pattern allows you to achieve this

Pattern: (?=\S+@example\.com)\S+

In this pattern, the look-ahead condition (?=\S+@example\.com) asserts that the
following characters
must match the pattern \S+@example\.com, which represents an email address ending
with the domain "example.com".

The non-consuming behavior of look-ahead conditions in regex patterns is valuable
in situations where you
need to match a specific pattern followed by another pattern
without including the latter in the match, allowing you to precisely control the
results of your program.

```

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

positive look-ahead : This is a positive assertion that matches the current position only if the specified pattern ahead of it matches.

negative look-ahead: This is a negative assertion that matches the current position only if the specified pattern ahead of it does not match

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

```
using named groups in regular expressions improves readability, enhances the self-documentation of patterns,
provides flexibility
```

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

```
Yes,by combining named groups with backreferences, you can identify repeated items within
a target string and perform further processing or analysis based on those repeated items.
```

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?

In [ ]:

```
When parsing a string, the Scanner interface in Python provides a convenient way to
string, separating it into individual elements or tokens when we use re.findall()
```

Q10. Does a scanner object have to be named scanner

In [ ]:

```
No, a scanner object does not have to be named "scanner"
```