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 mode tries to match the longest possible term first and non-greedy tries to match the shortest possible term first.

lets take following example : Mool Chand is a good human being. Mool Chand always pays tax. Mool Chand is learning Python.

non greedy : ".\*?" for ex : Mool Chand.\*? Mool Chand only pass string up to starting of second statement. Where as greedy : ".\*" Mool Chand .\* Mool Chand pass string up to thrid statement.

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:** In above question example, if there is only one statement “Mool Chand is a good human being.”, then there will be no difference either I use greedy or non greedy pattern.

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:** No, use of nontagged group will not make any difference in this case.

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

**Ans:** Consider the following example, which is using a nontagged category that would have a significant impact on the program’s outcomes:

http://example.com/

<https://example.com/questions/tagged/regex>

Now, if I apply the regex below over it...

(https?|ftp)://([^/\r\n]+)(/[^\r\n]\*)?

Match "http://example.com/"

Group 1: "http"

Group 2: "example.com"

Group 3: "/"

Match "https://example.com/questions/tagged/regex"

Group 1: "https"

Group 2: "example.com"

Group 3: "/questions/tagged/regex"

But I don't care about the protocol -- I just want the host and path of the URL. So, I change the regex to include the non-capturing group (?:).

(?:https?|ftp)://([^/\r\n]+)(/[^\r\n]\*)?

Now, my result looks like this:

Match "http://example.com/"

Group 1: "example.com"

Group 2: "/"

Match "https://example.com/questions/tagged/regex"

Group 1: "example.com"

Group 2: "/questions/tagged/regex"

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:** Lookahead is used as an assertion in Python regular expressions to determine success or failure whether the pattern is ahead i.e to the right of the parser’s current position. They don’t match anything. Hence, they are called as zero-width assertions.

e.g

import re

example = re.search(r'hello(?=[a-z])', "helloguys")

print("Pattern:", example.group())

it will give output “hello”, lookahead assertion (?=[a-z]) specifies that what follows geeks must be a lowercase alphabetic character. In this case, it’s the character g, a match is found.

import re

example = re.search(r'hello(?=[a-z])',

"hello123")

print(example)

it will give output “None”, since the next character is 1.

Similarly, if we use normal regex pattern, this is how the output is differing:

import re

example2 = re.search(r'hello([a-z])',

"helloguys")

print(example2.group())

it will give output “hellog”

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

**Ans:**

Positive lookahead, is to assure that the search string is followed by <lookahead\_regex>.

Syntax:

(?=<lookahead\_regex>)

Negative lookahead is opposite of lookahead. It is to assure that the search string is not followed by <lookahead\_regex>.

Syntax:

(?!<lookahead\_regex>)

e.g.

import re

example1 = re.search('hello(?=[a-z])',

'helloguys')

print('Positive Lookahead:', example1.group())

example2 = re.search('hello(?![a-z])',

'hello123')

print('Negative Lookahead:', example2.group())

Positive Lookahead: hello

Negative Lookahead: hello

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

**Ans:** Named groups are handy because they let you use easily-remembered names, instead of having to remember numbers.

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

**Ans:**

re.compile(r'\b(?P<word>\w+)\s+(?P=word)\b')

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:**

Example of scanner feature is below:

import sre

def s\_ident(scanner, token): return token

def s\_operator(scanner, token): return "op%s" % token

def s\_float(scanner, token): return float(token)

def s\_int(scanner, token): return int(token)

scanner = sre.Scanner([

(r"[a-zA-Z\_]\w\*", s\_ident),

(r"\d+\.\d\*", s\_float),

(r"\d+", s\_int),

(r"=|\+|-|\\*|/", s\_operator),

(r"\s+", None),

])

print scanner.scan("sum = 3\*foo + 312.50 + bar")

output of this program will be - (['sum', 'op=', 3, 'op\*', 'foo', 'op+', 312.5, 'op+', 'bar'], '')

difference between scanner and findall is this, we can provide multiple regular expressions to scan in scanner object.

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

**Ans:** yes, it should be named to use function names for handling the found tokens.