Python Advanced Assignment-17

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 character can you introduce or change?

Ans.

- Greedy: Matches as much as possible.

- Example: `a.*b` on "aabcab" matches "aabcab".

- Non-greedy: Matches as little as possible.

- Example: `a.*?b` on "aabcab" matches "aab".

Minimum effort to make a greedy pattern non-greedy:

- Introduce `?` after the quantifier (`*`, `+`, `?`, `{}`).

Example of transformation:

- Greedy: `.*`

- Non-greedy: `.*?`

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 vs non-greedy matters when multiple possible matches exist for a pattern. A greedy pattern will try to match the longest possible sequence, while a non-greedy pattern will match the shortest possible sequence.

- Example:

- Pattern: `a.*b`

- String: "aabcab"

- Greedy match: "aabcab"

- Non-greedy match: "aab"

If you're looking for a non-greedy match but the only available one is greedy, you can transform the greedy pattern to non-greedy by adding `?` after the quantifier.

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 non-tagged group likely to make any practical difference?

Ans.

In a simple match scenario, non-tagged groups (using `(?:...)`) won't make much difference since you're only concerned with finding the match, not capturing or using the groups. However, if you're interested in capturing certain portions of the match, using regular tagged groups would be more practical.

Q4: Describe a scenario in which using a non-tagged category would have a significant impact on the program's outcomes.

Ans.

When you want to apply grouping without capturing—such as when you're interested in repeating a pattern but don't need to reference the group later—you would use a non-tagged group (`(?:...)`).

Example:

- Regex to match two repetitions of "abc" without capturing:

```
  re.search(r'(?:abc){2}', 'abcabc')  # Matches without capturing
```

Using non-tagged groups can also prevent unnecessary indexing of groups, which reduces memory consumption and makes the regex more efficient.

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 program.

Ans.

Look-ahead is useful when you need to check for a condition without advancing the search pointer, so the following regex still includes the characters you looked ahead for.

Example:

- Match a word followed by a digit without consuming the digit:

```
  re.search(r'\w+(?=\d)', 'abc123')  # Matches 'abc', leaves '123'
```

In this case, the digit "1" is not consumed, allowing subsequent regex operations to act on it.

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

Ans.

- Positive look-ahead (`(?=...)`): Ensures that the pattern ahead exists.

- Example: `\w+(?=\d)` matches a word that is followed by a digit (but does not consume the digit).

- Negative look-ahead (`(?!...)`): Ensures that the pattern ahead does not exist.

- Example: `\w+(?!\d)` matches a word that is not followed by a digit.

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 using `(?P<name>...)` makes your code more readable and maintainable. Instead of remembering group numbers, you can use meaningful names for each group, which clarifies the intent and structure of the regex.

Example:

```
match = re.search(r'(?P<word>\w+)\s(?P<number>\d+)', 'abc 123')

print(match.group('word'))  # Outputs: 'abc'

print(match.group('number'))  # Outputs: '123'
```

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

Ans.

Yes, named groups can be used to match repeated patterns. For example, to find repeated words in a string:

Example:

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

re.search(pattern, 'The cow cow jumped over the moon')  # Matches 'cow cow'
```

This regex finds two adjacent occurrences of the same word.

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 allows you to tokenize the string and apply a function to each match, which `re.findall()` does not provide. With `Scanner`, you can process matches incrementally and react to each match with custom logic.

Example:

```
import re

scanner = re.Scanner([

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

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

])

tokens, remainder = scanner.scan("123 abc 456 def")

print(tokens)  # Outputs: [('NUMBER', '123'), ('WORD', 'abc'), ('NUMBER', '456'), ('WORD', 'def')]
```

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

Ans.

No, the scanner object does not have to be named `scanner`. It can be named anything you like, just like any other variable in Python.

Example:

```
import re

my_scanner = re.Scanner([

    (r'\d+', lambda s, t: ("NUMBER", t)),

    (r'\w+', lambda s, t: ("WORD", t)),

])
```