Assignment 17 Solutions

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: The Main difference between the greedy and the non-greedy match is the following: The greedy match will try to match as many repetitions of the quantified pattern as possible. The non-greedy match will try to match as few repetitions of the quantified pattern as possible.

In [1]:

```
import re
print(re.findall("a*", "aaaaaa")) # Greedy Match Synatx
print(re.findall("a*?", "aaaaaa")) # Non Greddy Syntax
```

```
['aaaaaa', '']
['', 'a', '', 'a', '', 'a', '', 'a', '', 'a', '', 'a', '']
```

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: So the difference between the greedy and the non-greedy match is the following: The greedy match will try to match as many repetitions of the quantified pattern as possible. The non-greedy match will try to match as few repetitions of the quantified pattern as possible.

example: To make the quantifier non-greedy you simply follow it with a '?' the first 3 characters and then the following 'ab' is matched. greedy by appending a '?' symbol to them: *?, +?, ??, {n,m}?, and {n,}?.

if we're looking for a non-greedy match but the only one available is greedy that will result in matching the shortest possible string.

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: In this Case, the Non-Tagged Group will not make any difference. This is shown below:

In [2]:

```
import re
phoneNumRegex = re.compile(r'\d\d\d')
num = phoneNumRegex.search('My number is 983-451-7777.')
print(f'Phone number found -> {num.group()}') # Non Tagged group
print(f'Phone number found -> {num.group(0)}') # Tagged Group
```

```
Phone number found -> 983
Phone number found -> 983
```

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

Ans: Here in the below Code Snippet . decimal is not tagged or captured. Hence, it will be useful in scenarios where the separator of value in a string is of no use and we need to capture only the values

In [3]:

```
import re

# \d is equivalent to [0-9].
p = re.compile('\d')
print(p.findall("I went to him at 11 A.M. on 4th July 1886"))

# \d+ will match a group on [0-9], group of one or greater size
p = re.compile('\d+')
print(p.findall("I went to him at 11 A.M. on 4th July 1886"))

#a simple + operator can make a huge difference in outcome in case of regular expressions
```

```
['1', '1', '4', '1', '8', '8', '6']
['11', '4', '1886']
```

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: While the order of lookaheads doesn't matter on a logical level, we should keep in mind that it may matter for matching speed. If one lookahead is more likely to fail than the other two, it makes little sense to place it in third position and expend a lot of energy checking the first two conditions. Make it first, so that if we're going to fail, we fail early—an application of the design to fail principle from the regex style guide.

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

Ans: Positive Look-ahead allows to add a condition for what follows. Negative Lookahead is similar, but it looks behind. That is, it allows to match a pattern only if there's something before it.

Syntax Positive Look-Ahead: X(?=Y) Syntax Negative Look-Ahead: X(?!Y)

In [5]:

```
#Positive Look Ahead
import re

s = '1 Python is about 4 feet long'
pattern = '\d+(?=\s*feet)'

matches = re.finditer(pattern,s)
for match in matches:
    print(match.group())
```

```
4
```

In [6]:

```
#Negative Look Ahead
import re

s = '1 Python is about 4 feet long'
pattern = '\d+(?!\s*feet)'

matches = re.finditer(pattern,s)
for match in matches:
    print(match.group())
```

```
1
```

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

Ans: The advantage to named groups is that it adds readability and understandability to the code, so that you can easily see what part of a regular expression match is being referenced

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

In [7]:

```
import re
text = "The cow jumped over the moon"
regobj=re.compile(r'(?P<w1>The)',re.I)
regobj.findall(text)
```

Out[7]:

```
['The', 'the']
```

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: re.findall() module is used to search for all occurrences that match a given pattern. In contrast, re.search() will only return the first occurrence that matches the specified pattern. re.findall() will iterate over all the lines of the file and will return all non-overlapping matches of pattern in a single step.

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

Ans: The scan module is part of the microscopy package. It provides a framework for multidimensional scanning routines while acquiring data. Not madatory to keep the same name.