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?

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.

A greedy quantifier such as ?, \*, +, and {m,n} matches as many characters as possible (longest match). For example, the regex 'a+' will match as many 'a's as possible in your string 'aaaa'—even though the substrings 'a', 'aa', 'aaa' all match the regex 'a+'.

A non-greedy quantifier such as ??, \*?, +?, and {m,n}? matches as few characters as possible (shortest possible match). For example, the regex 'a+?' will match as few 'a's as possible in your string 'aaaa'. Thus, it matches the first character 'a' and is done with it.

You can make the default quantifiers ? , \* , + , {m} , and {m,n} non-greedy by appending a question mark symbol '?' to them: ?? , \*? , +? , and {m,n}? . they “consume” or match as few characters as possible so that the regex pattern is still satisfied.

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?

It becomes important if you are trying to match certain parts of an expression. Sometimes you don't want to match everything - as little as possible. Sometimes you want to match as much as possible. Nothing more to it.

Greedy Quantifiers are like the IRS

They’ll take as much as they can. e.g. matches with this regex: .\*

$50,000

Bye-bye bank balance.

Non-greedy quantifiers - they take as little as they can

Ask for a tax refund: the IRS sudden becomes non-greedy - and return as little as possible: i.e. they use this quantifier:

(.{2,5}?)([0-9]\*) against this input: $50,000

The first group is non-needy and only matches $5 – so I get a $5 refund against the $50,000 input.

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?

Sometimes, you may want to create a group but don't want to capture it in the groups of the match. To do that, you can use a non-capturing group with the following syntax: (?:X)

The main benefit of non-capturing groups is that you can add them to a regex without upsetting the numbering of the capturing groups in the regex. They also offer (slightly) better performance as the regex engine doesn't have to keep track of the text matched by non-capturing groups.

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

Sometimes you want to use parentheses to group parts of an expression together, but you don't want the group to capture anything from the substring it matches. To do this use (?: and ) to enclose the group.

You might want this in order to make it easier to keep track of backreferences. For example,

**(?:\$|(?:USD))([0-9]+)\.([0-9]{2})**

matches dollar amounts like **$10.43** and **USD19.98** and saves the dollar amount in **\1** and the cents amount in **\2**. Now a Java program can refer to the dollar and cents amounts as group 1 and group 2, rather than more obscure numbers. (See chapter 11 for how Java programs can use the values held in backreferences.)

Another benefit of non-capturing groups is that matching is performed faster.

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.

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.

# importing regex

import re

# lookahead example

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

# display output

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

print("Pattern found from index:",

example.start(), "to",

example.end())

Sometimes, you want to match X but only if it is followed by Y. In this case, you can use the lookahead in regular expressions.

The syntax of the lookahead is as follows:

X(?=Y)

This syntax means to search for X but matches only if it is followed by Y.

For example, suppose you have the following string:

'1 Python is about 4 feet long'

And you want to match the number (4) that is followed by a space and the literal string feet, not the number 1. In this case, you can use the following pattern that contains a lookahead:

\d+(?=\s\*feet)

In this pattern:

* \d+ is the combination of the digit character set with the + quantifier that matches one or more digits.
* ?= is the lookahead syntax
* \s\* is the combination of the whitespace character set and \* quantifier that matches zero or more whitespaces.
* feet matches the literal string feet.

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

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.

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

# import required module

import re

# positive lookahead

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

'geeksforgeeks')

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

# negative lookahead

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

'geeks123')

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

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

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.

Nearly all modern regular expression engines support numbered capturing groups and numbered backreferences. Long regular expressions with lots of groups and backreferences may be hard to read. They can be particularly difficult to maintain as adding or removing a capturing group in the middle of the regex upsets the numbers of all the groups that follow the added or removed group.

Python’s re module was the first to offer a solution: named capturing groups and named backreferences. (?P<name>group) captures the match of group into the backreference “name”. name must be an alphanumeric sequence starting with a letter. group can be any regular expression. You can reference the contents of the group with the named backreference (?P=name). The question mark, P, angle brackets, and equals signs are all part of the syntax. Though the syntax for the named backreference uses parentheses, it’s just a backreference that doesn’t do any capturing or grouping. The HTML tags example can be written as <(?P<tag>[A-Z][A-Z0-9]\*)\b[^>]\*>.\*?</(?P=tag)>.

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

One of the most powerful regular-expression capabilities is to selectively search-and-replace patterns within a string of text. Here’s one possible use (out of zillions): to transform a target string by replacing each repeated pair of words with just one word.

For example, given this text:

The cow cow jumped over the the moon.

it would be useful to produce a string consisting of:

The cow jumped over the moon.

The regex\_replace function performs this task by returning the transformed string. It has the following syntax:

regex\_replace(target\_string, regex\_obj, replacement\_pattern\_str);

The replacement\_pattern\_str is a string that can contain the following special sequences (in addition to ordinary characters).

Censoring a text string using a dictionary and replacing words:

import re

blacklist = ['ccc', 'eee']

def replace(match):

word = match.group()

if word.lower() in blacklist:

return 'x' \* len(word)

else:

return word

text = 'aaa bbb ccc. ddd eee xcccx.'

text = re.sub(r'\b\w\*\b', replace, text, flags=re.I|re.U)

print(text)

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?

Enter The Scanner:

This is where things get interesting. For the last 15 years or so, there has been a completely undocumented feature in the regular expression engine: the scanner. The scanner is a property of the underlying SRE pattern object where the engine keeps matching after it found a match for the next one. There even exists an re.Scanner class (also undocumented) which is built on top of the SRE pattern scanner which gives this a slightly higher level interface.

The scanner as it exists in the re module is not very useful unfortunately for making the 'not matching' part faster, but looking at its sourcecode reveals how it's implemented: on top of the SRE primitives.

The way it works is it accepts a list of regular expression and callback tuples. For each match it invokes the callback with the match object and then builds a result list out of it. When we look at how it's implemented it manually creates SRE pattern and subpattern objects internally. (Basically it builds a larger regular expression without having to parse it). Armed with this knowledge we can extend this:

from sre\_parse import Pattern, SubPattern, parse

from sre\_compile import compile as sre\_compile

from sre\_constants import BRANCH, SUBPATTERN

class Scanner(object):

def \_\_init\_\_(self, rules, flags=0):

pattern = Pattern()

pattern.flags = flags

pattern.groups = len(rules) + 1

self.rules = [name for name, \_ in rules]

self.\_scanner = sre\_compile(SubPattern(pattern, [

(BRANCH, (None, [SubPattern(pattern, [

(SUBPATTERN, (group, parse(regex, flags, pattern))),

]) for group, (\_, regex) in enumerate(rules, 1)]))

])).scanner

def scan(self, string, skip=False):

sc = self.\_scanner(string)

match = None

for match in iter(sc.search if skip else sc.match, None):

yield self.rules[match.lastindex - 1], match

if not skip and not match or match.end() < len(string):

raise EOFError(match.end())

So how do we use this? Like this:

scanner = Scanner([

('whitespace', r'\s+'),

('plus', r'\+'),

('minus', r'\-'),

('mult', r'\\*'),

('div', r'/'),

('num', r'\d+'),

('paren\_open', r'\('),

('paren\_close', r'\)'),

])

for token, match in scanner.scan('(1 + 2) \* 3'):

print (token, match.group())

In this form it will raise an EOFError in case it cannot lex something, but if you pass skip=True then it skips over unlexable parts which is perfect for building things like wiki syntax lexers.

Scanning with Holes:

When we skip, we can use match.start() and match.end() to figure out which parts we skipped over. So here the first example adjusted to do exactly that:

scanner = Scanner([

('bold', r'\\*\\*'),

('link', r'\[\[(.\*?)\]\]'),

])

def tokenize(string):

pos = 0

for rule, match in self.scan(string, skip=True):

hole = string[pos:match.start()]

if hole:

yield 'text', hole

yield rule, match.group()

pos = match.end()

hole = string[pos:]

if hole:

yield 'text', hole

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

No, the object does not need to be named scanner.