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:

Definition Greedy Quantifier:

**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+'**.**

Definition Non-Greedy Quantifier:

**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.**

## Python Regex Quantifiers

The word “[quantifier](https://www.merriam-webster.com/dictionary/quantity)” originates from latin: it’s meaning is **quantus = how much / how often**.

**This is precisely what a regular expression quantifier means: you tell the regex engine how often you want to match a given pattern.**

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 matching is the default behavior of regular expressions, where the regular expression engine will try to match as much text as possible. In contrast, non-greedy matching, also known as lazy matching, tries to match as little text 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 a nontagged group likely to make any practical difference?

ANS:

On an abstract level a regular expression, regex for short, is a shorthand representation for a set. A set of strings.

Say we have a list of all valid zip codes. Instead of keeping that long and unwieldy list around, it’s often more practical to have a short and precise pattern that completely describes that set. Whenever you want to check whether a string is a valid zip code, you can match it against the pattern. You’ll get a true or false result indicating whether the string belongs to the set of zip codes the regex pattern represents.

# The Building Blocks of a Regular Expression

A regular expression pattern is constructed from distinct building blocks. It may contain [literals](https://towardsdatascience.com/everything-you-need-to-know-about-regular-expressions-8f622fe10b03#literals), [character classes](https://towardsdatascience.com/everything-you-need-to-know-about-regular-expressions-8f622fe10b03#character-classes), [boundary matchers](https://towardsdatascience.com/everything-you-need-to-know-about-regular-expressions-8f622fe10b03#boundary-matchers), [quantifiers](https://towardsdatascience.com/everything-you-need-to-know-about-regular-expressions-8f622fe10b03#quantifiers), [groups](https://towardsdatascience.com/everything-you-need-to-know-about-regular-expressions-8f622fe10b03#groups) and the [OR operator](https://towardsdatascience.com/everything-you-need-to-know-about-regular-expressions-8f622fe10b03#the-or-operator).

Let’s dive in and look at some examples.

## Literals

The most basic building block in a regular expression is a character a.k.a. literal. Most characters in a regex pattern do not have a special meaning, they simply match themselves. Consider the following pattern:

I am a harmless regex pattern

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

ANS:

A POS tagger considers surrounding words while assigning a tag. For example, the previous sentence, “Book the flight”, will become a list of each word with its corresponding POS tag – [(“Book”, “Verb”), (“the”, “Det”), (“flight”, “Noun”)].

One way to address the trade-off between accuracy and coverage is to use the more accurate algorithms when we can, but to fall back on algorithms with wider coverage when necessary. For example, we could combine the results of a bigram tagger, a unigram tagger, and a default tagger, as follows:

1. Try tagging the token with the bigram tagger.
2. If the bigram tagger is unable to find a tag for the token, try the unigram tagger.
3. If the unigram tagger is also unable to find a tag, use a default tagger.

Most NLTK taggers permit a backoff-tagger to be specified. The backoff-tagger may itself have a backoff tagger:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | >>> t0 = nltk.DefaultTagger('NN')  >>> t1 = nltk.UnigramTagger(train\_sents, backoff=t0)  >>> t2 = nltk.BigramTagger(train\_sents, backoff=t1)  >>> t2.evaluate(test\_sents)  0.844513... | |

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:

The engine starts with the lookbehind and the first character in the string. In this case, the lookbehind tells the engine to step back one character, and see if a can be matched there. The engine cannot step back one character because there are no characters before the t.

Lookarounds often cause confusion to the regex apprentice. I believe this confusion promptly disappears if one simple point is firmly grasped. It is that at the end of a lookahead or a lookbehind, the regex engine hasn't moved on the string. You can chain three more lookaheads after the first, and the regex engine still won't move. In fact, that's a useful technique.  
  
A quick syntax reminder  
This page digs deep into the details of lookahead and lookbehind and assumes you've already become familiar with the basic syntax, perhaps by reading the [lookaround section](https://www.rexegg.com/regex-disambiguation.html" \l "lookarounds) of the reference on (? … ) syntax. As a quick reminder before we dive in, here are the four lookarounds.

|  |  |  |
| --- | --- | --- |
| Lookaround | Name | What it Does |
| (?=foo) | Lookahead | Asserts that what immediately follows the current position in the string is foo |
| (?<=foo) | Lookbehind | Asserts that what immediately precedes the current position in the string is foo |
| (?!foo) | Negative Lookahead | Asserts that what immediately follows the current position in the string is not foo |
| (?<!foo) | Negative Lookbehind | Asserts that what immediately precedes the current position in the string is not foo |

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

ANS:

Positive lookahead: (?= «pattern») matches if pattern matches what comes after the current location in the input string. Negative lookahead: (?! «pattern») matches if pattern does not match what comes after the current location in the input string.

The negative lookahead construct is the pair of parentheses, with the opening parenthesis followed by a question mark and an exclamation point. Inside the lookahead, we have the trivial regex u. Positive lookahead works just the same. q(?=u) matches a q that is followed by a u, without making the u part of the match.

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.

* Regular expression helps the programmers to validate text string.
* It offers a powerful tool to analyze and search a pattern as well as to modify the text string.
* By using regexes functions, simple and easy solutions are provided to identify the patterns.

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

ANS:

## Replacing Text

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**);**

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:

A [Regular expression](https://www.geeksforgeeks.org/regular-expression-python-examples-set-1/) (sometimes called a Rational expression) is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. “find and replace”-like operations. Regular expressions are a generalized way to match patterns with sequences of characters.

Module Regular Expressions (RE) specifies a set of strings (pattern) that matches it. To understand the RE analogy, MetaCharacters are useful, important and will be used in functions of module re.

There are a total of 14 metacharacters and will be discussed as they follow into functions:

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

ANS:

nce we import the package, here is how we can create Scanner objects. // read input from the input stream Scanner sc1 = new Scanner(InputStream input); // read input from files Scanner sc2 = new Scanner(File file); // read input from a string Scanner sc3 = new Scanner(String str);