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?

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 Greedy Match

A greedy match means that the regex engine (the one which tries to find your pattern in the string) matches as many characters as possible.

For example, the regex 'a+' will match as many 'a's as possible in your string 'aaaa'. Although the substrings 'a', 'aa', 'aaa' all match the regex 'a+', it’s not enough for the regex engine. It’s always hungry and tries to match even more.

In other words, the greedy quantifiers give you the longest match from a given position in the string.

As it turns out, all default quantifiers ?, \*, +, {m}, and {m,n} you’ve learned above are greedy: they “consume” or match as many characters as possible so that the regex pattern is still satisfied.

Here are the above examples again that all show how greedy the regex engine is:

>>> import re

>>> re.findall('a?', 'aaaa')

['a', 'a', 'a', 'a', '']

>>> re.findall('a\*', 'aaaa')

['aaaa', '']

>>> re.findall('a+', 'aaaa')

['aaaa']

>>> re.findall('a{3}', 'aaaa')

['aaa']

>>> re.findall('a{1,2}', 'aaaa')

['aa', 'aa']

In all cases, a shorter match would also be valid. But as the regex engine is greedy per default, those are not enough for the regex engine

Python Regex Non-Greedy Match

A non-greedy match means that the regex engine matches as few characters as possible—so that it still can match the pattern in the given string.

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. Then, it moves on to the second character (which is also a match) and so on.

In other words, the non-greedy quantifiers give you the shortest possible match from a given position in the string.

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?

Greedy and lazy quantifiers

Quantifiers are very simple from the first sight, but in fact they can be tricky.

We should understand how the search works very well if we plan to look for something more complex than /\d+/.

Let’s take the following task as an example.

We have a text and need to replace all quotes "..." with guillemet marks: «...». They are preferred for typography in many countries.

For instance: "Hello, world" should become «Hello, world». There exist other quotes, such as „Witam, świat!” (Polish) or 「你好，世界」 (Chinese), but for our task let’s choose «...».

The first thing to do is to locate quoted strings, and then we can replace them.

A regular expression like /".+"/g (a quote, then something, then the other quote) may seem like a good fit, but it isn’t!

Let’s try it:

let regexp = /".+"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(regexp) ); // "witch" and her "broom"

…We can see that it works not as intended!

Instead of finding two matches "witch" and "broom", it finds one: "witch" and her "broom".

That can be described as “greediness is the cause of all evil”.

[Greedy search](https://javascript.info/regexp-greedy-and-lazy" \l "greedy-search)

To find a match, the regular expression engine uses the following algorithm:

For every position in the string

Try to match the pattern at that position.

If there’s no match, go to the next position.

These common words do not make it obvious why the regexp fails, so let’s elaborate how the search works for the pattern ".+".

The first pattern character is a quote ".

The regular expression engine tries to find it at the zero position of the source string a "witch" and her "broom" is one, but there’s a there, so there’s immediately no match.

Then it advances: goes to the next positions in the source string and tries to find the first character of the pattern there, fails again, and finally finds the quote at the 3rd position:

The quote is detected, and then the engine tries to find a match for the rest of the pattern. It tries to see if the rest of the subject string conforms to .+".

In our case the next pattern character is . (a dot). It denotes “any character except a newline”, so the next string letter 'w' fits:

Then the dot repeats because of the quantifier .+. The regular expression engine adds to the match one character after another.

…Until when? All characters match the dot, so it only stops when it reaches the end of the string:

Now the engine finished repeating .+ and tries to find the next character of the pattern. It’s the quote ". But there’s a problem: the string has finished, there are no more characters!

The regular expression engine understands that it took too many .+ and starts to backtrack.

In other words, it shortens the match for the quantifier by one character:

Now it assumes that .+ ends one character before the string end and tries to match the rest of the pattern from that position.

If there were a quote there, then the search would end, but the last character is 'e', so there’s no match.

…So the engine decreases the number of repetitions of .+ by one more character:

The quote '"' does not match 'n'.

The engine keep backtracking: it decreases the count of repetition for '.' until the rest of the pattern (in our case '"') matches:

The match is complete.

So the first match is "witch" and her "broom". If the regular expression has flag g, then the search will continue from where the first match ends. There are no more quotes in the rest of the string is one, so no more results.

That’s probably not what we expected, but that’s how it works.

In the greedy mode (by default) a quantified character is repeated as many times as possible.

The regexp engine adds to the match as many characters as it can for .+, and then shortens that one by one, if the rest of the pattern doesn’t match.

For our task we want another thing. That’s where a lazy mode can help.

[Lazy mode](https://javascript.info/regexp-greedy-and-lazy" \l "lazy-mode)

The lazy mode of quantifiers is an opposite to the greedy mode. It means: “repeat minimal number of times”.

We can enable it by putting a question mark '?' after the quantifier, so that it becomes \*? or +? or even ?? for '?'.

To make things clear: usually a question mark ? is a quantifier by itself (zero or one), but if added after another quantifier (or even itself) it gets another meaning – it switches the matching mode from greedy to lazy.

The regexp /".+?"/g works as intended: it finds "witch" and "broom":

let regexp = /".+?"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(regexp) ); // "witch", "broom"

To clearly understand the change, let’s trace the search step by step.

The first step is the same: it finds the pattern start '"' at the 3rd position:

The next step is also similar: the engine finds a match for the dot '.':

And now the search goes differently. Because we have a lazy mode for +?, the engine doesn’t try to match a dot one more time, but stops and tries to match the rest of the pattern '"' right now:

If there were a quote there, then the search would end, but there’s 'i', so there’s no match.

Then the regular expression engine increases the number of repetitions for the dot and tries one more time:

Failure again. Then the number of repetitions is increased again and again…

…Till the match for the rest of the pattern is found:

The next search starts from the end of the current match and yield one more result:

In this example we saw how the lazy mode works for +?. Quantifiers \*? and ?? work the similar way – the regexp engine increases the number of repetitions only if the rest of the pattern can’t match on the given position.

Laziness is only enabled for the quantifier with ?.

Other quantifiers remain greedy.

For instance:

alert( "123 456".match(/\d+ \d+?/) ); // 123 4

The pattern \d+ tries to match as many digits as it can (greedy mode), so it finds 123 and stops, because the next character is a space ' '.

Then there’s a space in the pattern, it matches.

Then there’s \d+?. The quantifier is in lazy mode, so it finds one digit 4 and tries to check if the rest of the pattern matches from there.

…But there’s nothing in the pattern after \d+?.

The lazy mode doesn’t repeat anything without a need. The pattern finished, so we’re done. We have a match 123 4.

Optimizations

Modern regular expression engines can optimize internal algorithms to work faster. So they may work a bit differently from the described algorithm.

But to understand how regular expressions work and to build regular expressions, we don’t need to know about that. They are only used internally to optimize things.

Complex regular expressions are hard to optimize, so the search may work exactly as described as well.

[Alternative approach](https://javascript.info/regexp-greedy-and-lazy" \l "alternative-approach)

With regexps, there’s often more than one way to do the same thing.

In our case we can find quoted strings without lazy mode using the regexp "[^"]+":

let regexp = /"[^"]+"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(regexp) ); // "witch", "broom"

The regexp "[^"]+" gives correct results, because it looks for a quote '"' followed by one or more non-quotes [^"], and then the closing quote.

When the regexp engine looks for [^"]+ it stops the repetitions when it meets the closing quote, and we’re done.

Please note, that this logic does not replace lazy quantifiers!

It is just different. There are times when we need one or another.

Let’s see an example where lazy quantifiers fail and this variant works right.

For instance, we want to find links of the form <a href="..." class="doc">, with any href.

Which regular expression to use?

The first idea might be: /<a href=".\*" class="doc">/g.

Let’s check it:

let str = '...<a href="link" class="doc">...';

let regexp = /<a href=".\*" class="doc">/g;

// Works!

alert( str.match(regexp) ); // <a href="link" class="doc">

It worked. But let’s see what happens if there are many links in the text?

let str = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';

let regexp = /<a href=".\*" class="doc">/g;

// Whoops! Two links in one match!

alert( str.match(regexp) ); // <a href="link1" class="doc">... <a href="link2" class="doc">

Now the result is wrong for the same reason as our “witches” example. The quantifier .\* took too many characters.

The match looks like this:

<a href="....................................." class="doc">

<a href="link1" class="doc">... <a href="link2" class="doc">

Let’s modify the pattern by making the quantifier .\*? lazy:

let str = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';

let regexp = /<a href=".\*?" class="doc">/g;

// Works!

alert( str.match(regexp) ); // <a href="link1" class="doc">, <a href="link2" class="doc">

Now it seems to work, there are two matches:

<a href="....." class="doc"> <a href="....." class="doc">

<a href="link1" class="doc">... <a href="link2" class="doc">

…But let’s test it on one more text input:

let str = '...<a href="link1" class="wrong">... <p style="" class="doc">...';

let regexp = /<a href=".\*?" class="doc">/g;

// Wrong match!

alert( str.match(regexp) ); // <a href="link1" class="wrong">... <p style="" class="doc">

Now it fails. The match includes not just a link, but also a lot of text after it, including <p...>.

Why?

That’s what’s going on:

First the regexp finds a link start <a href=".

Then it looks for .\*?: takes one character (lazily!), check if there’s a match for " class="doc"> (none).

Then takes another character into .\*?, and so on… until it finally reaches " class="doc">.

But the problem is: that’s already beyond the link <a...>, in another tag <p>. Not what we want.

Here’s the picture of the match aligned with the text:

<a href="..................................." class="doc">

<a href="link1" class="wrong">... <p style="" class="doc">

So, we need the pattern to look for <a href="...something..." class="doc">, but both greedy and lazy variants have problems.

The correct variant can be: href="[^"]\*". It will take all characters inside the href attribute till the nearest quote, just what we need.

A working example:

let str1 = '...<a href="link1" class="wrong">... <p style="" class="doc">...';

let str2 = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';

let regexp = /<a href="[^"]\*" class="doc">/g;

// Works!

alert( str1.match(regexp) ); // null, no matches, that's correct

alert( str2.match(regexp) ); // <a href="link1" class="doc">, <a href="link2" class="doc">

[Summary](https://javascript.info/regexp-greedy-and-lazy" \l "summary)

Quantifiers have two modes of work:

Greedy

By default the regular expression engine tries to repeat the quantified character as many times as possible. For instance, \d+ consumes all possible digits. When it becomes impossible to consume more (no more digits or string end), then it continues to match the rest of the pattern. If there’s no match then it decreases the number of repetitions (backtracks) and tries again.

Lazy

Enabled by the question mark ? after the quantifier. The regexp engine tries to match the rest of the pattern before each repetition of the quantified character.

As we’ve seen, the lazy mode is not a “panacea” from the greedy search. An alternative is a “fine-tuned” greedy search, with exclusions, as in the pattern "[^"]+".

[Tasks](https://javascript.info/regexp-greedy-and-lazy#tasks)

[A match for /d+? d+?/](https://javascript.info/regexp-greedy-and-lazy" \l "a-match-for-d-d)

What’s the match here?

alert( "123 456".match(/\d+? \d+?/g) ); // ?

solution

[Find HTML comments](https://javascript.info/regexp-greedy-and-lazy" \l "find-html-comments)

Find all HTML comments in the text:

let regexp = /your regexp/g;

let str = `... <!-- My -- comment

test --> .. <!----> ..

`;

alert( str.match(regexp) ); // '<!-- My -- comment \n test -->', '<!---->'

solution

[Find HTML tags](https://javascript.info/regexp-greedy-and-lazy" \l "find-html-tags)

Create a regular expression to find all (opening and closing) HTML tags with their attributes.

An example of use:

let regexp = /your regexp/g;

let str = '<> <a href="/"> <input type="radio" checked> <b>';

alert( str.match(regexp) ); // '<a href="/">', '<input type="radio" checked>', '<b>'

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?

Input:

string - GraphTreesGraph

pattern - aba

Output:

a->Graph

b->Trees

Input:

string - GraphGraphGraph

pattern - aaa

Output:

a->Graph

Input:

string - GeeksforGeeks

pattern - GfG

Output:

G->Geeks

f->for

Input:

string - GeeksforGeeks

pattern - GG

Output:

No solution exists

[Recommended: Please try your approach on {IDE} first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

We can solve this problem with the help of Backtracking. For each character in the pattern, if the character is not seen before, we consider all possible sub-strings and recurse to see if it leads to the solution or not. We maintain a map that stores sub-string mapped to a pattern character. If pattern character is seen before, we use the same sub-string present in the map. If we found a solution, for each distinct character in the pattern, we print string mapped to it using our map.  
Below is C++ implementation of above idea –

CPP

Java

Python3

Javascript

|  |
| --- |
| // C++ program to find out if string follows  // a given pattern or not  #include <bits/stdc++.h>  using namespace std;    /\*  Function to find out if string follows a given      pattern or not        str - given string      n - length of given string      i - current index in input string      pat - given pattern      m - length of given pattern      j - current index in given pattern      map - stores mapping between pattern and string \*/  bool patternMatchUtil(string str, int n, int i,                      string pat, int m, int j,                      unordered\_map<char, string>& map)  {      // If both string and pattern reach their end      if (i == n && j == m)          return true;        // If either string or pattern reach their end      if (i == n || j == m)          return false;        // read next character from the pattern      char ch = pat[j];        // if character is seen before      if (map.find(ch)!= map.end())      {          string s = map[ch];          int len = s.size();            // consider next len characters of str          string subStr = str.substr(i, len);            // if next len characters of string str          // don't match with s, return false          if (subStr.compare(s))              return false;            // if it matches, recurse for remaining characters          return patternMatchUtil(str, n, i + len, pat, m,                                              j + 1, map);      }        // If character is seen for first time, try out all      // remaining characters in the string      for (int len = 1; len <= n - i; len++)      {          // consider substring that starts at position i          // and spans len characters and add it to map          map[ch] = str.substr(i, len);            // see if it leads to the solution          if (patternMatchUtil(str, n, i + len, pat, m,                                            j + 1, map))              return true;            // if not, remove ch from the map          map.erase(ch);      }        return false;  }    // A wrapper over patternMatchUtil()function  bool patternMatch(string str, string pat, int n, int m)  {      if (n < m)      return false;        // create an empty hashmap      unordered\_map<char, string> map;        // store result in a boolean variable res      bool res = patternMatchUtil(str, n, 0, pat, m, 0, map);        // if solution exists, print the mappings      for (auto it = map.begin(); res && it != map.end(); it++)          cout << it->first << "->" << it->second << endl;        // return result      return res;  }    // Driver code  int main()  {      string str = "GeeksforGeeks", pat = "GfG";        int n = str.size(), m = pat.size();        if (!patternMatch(str, pat, n, m))          cout << "No Solution exists";        return 0;  } |

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

What Are Learning Outcomes?

The learning outcomes provide an overview of what students should know, be able to do, or be able to value after completing a course or program. Learning outcomes means how much knowledge or skills a student should acquire through various assignments, course, class or programs, by the end of a specific time period. They are observable and measurable by knowledge, skills, abilities, values, etc. Learning outcomes describe clearly what it is you want your students to be able to do by the end of a course.

The learning outcomes focus more on student performance rather than they do on traditional techniques or courses.

However, in a broader perspective, the term “Learning Outcomes” can be defined as an amalgamation of a learner’s knowledge, skill set, and the ability to leverage them in real-time situations. There are several taxonomies such as SOLO (Structure of Observed Learning Outcomes), [Bloom’s Taxonomy](https://www.iitms.co.in/blog/blooms-taxonomy-importance-applications.html), Fink's Taxonomy, Wiggins and McTighe Taxonomy on education. An ideal learning outcome would be something that abides by all the aspects of Bloom’s Taxonomy as it promotes knowledge & skill development in individuals and makes them life-long learners.

What Is The Focus Of Student Learning Outcome?

The main focus of student learning outcomes is to make students academically sound, skillful, and prepare them for life-long learning.

When we speak of “learning outcomes” – we must pay attention to whether they are–

S-Specific

M-Measurable

A-Achievable

R-Realistic

T-Timely

In short, the learning outcomes must be “SMART" & clearly defined in terms of attainability! It will empower students to achieve the outcomes smoothly.

How To Write Learning Outcomes?

Learning goals must be articulated in such a way that they cover -

All the educational goals of the students as well as institutions

Offer a roadmap to achieve course outcomes & program outcomes

Based on knowledge & skill development & can be attained via teaching-learning methods

Focuses on students’ growth & enable them to develop a lifelong learner mindset

Makes students capable of handling problems in real-life situations

Practices To Avoid While Building Learning Outcomes

While articulating learning outcomes, the faculty or educators must take care of the following-

Do not focus on “teacher-centric” practices -always make the learning “student-centric”

Do not confine the learning outcomes to course or program outcomes

Do not include projects that are not aligned with the students learning outcomes

Do not force students to learn the theoretical concepts for an academic score, instead encourage them to learn with practical experiments

Be specific in terms of what to expect- the learning outcomes must be measurable

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.

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.

Jumping Points  
For easy navigation, here are some jumping points to various sections of the page:  
  
✽ [Lookahead Example: Simple Password Validation](https://www.rexegg.com/regex-lookarounds.html#password)  
✽ [The Order of Lookaheads Doesn't Matter… Almost](https://www.rexegg.com/regex-lookarounds.html#order)  
✽ [Lookarounds Stand their Ground](https://www.rexegg.com/regex-lookarounds.html" \l "stand_their_ground)  
✽ [Various Uses for Lookarounds](https://www.rexegg.com/regex-lookarounds.html#uses)  
✽ [Zero-Width Matches](https://www.rexegg.com/regex-lookarounds.html#zero_width)  
✽ [Positioning the Lookaround Before or After the Characters to be Matched](https://www.rexegg.com/regex-lookarounds.html#position)  
✽ [Lookarounds that Look on Both Sides: Back to the Future](https://www.rexegg.com/regex-lookarounds.html" \l "back_to_the_future)  
✽ [Compound Lookahead and Compound Lookbehind](https://www.rexegg.com/regex-lookarounds.html#compound)  
✽ [The Engine Doesn't Backtrack into Lookarounds (They're Atomic)](https://www.rexegg.com/regex-lookarounds.html#atomic)  
✽ [Fixed-Width, Constrained-Width and Infinite-Width Lookbehind](https://www.rexegg.com/regex-lookarounds.html#width)  
✽ [Lookarounds (Usually) Want to be Anchored](https://www.rexegg.com/regex-lookarounds.html" \l "anchor)

Lookahead Example: Simple Password Validation

Let's get our feet wet right away with an expression that validates a password. The technique shown here will be useful for all kinds of other data you might want to validate (such as email addresses or phone numbers).  
Our password must meet four conditions:  
  
1. The password must have between six and ten word characters \w  
2. It must include at least one lowercase character [a-z]  
3. It must include at least three uppercase characters [A-Z]  
4. It must include at least one digit \d  
  
We'll assume we're working in a regex flavor where \d only matches ASCII digits 0 through 9, unlike .NET and Python where that token can match any Unicode digit.

With lookarounds, your feet stay planted on the string. You're just looking, not moving!

Our initial strategy (which we'll later tweak) will be to stand at the beginning of the string and look ahead four times—once for each condition. We'll look to check we have the right number of characters, then we'll look for a lowercase letter, and so on. If all the lookaheads are successful, we'll know the string is a valid password… And we'll simply gobble it all up with a plain .\*  
  
Let's start with condition 1  
A string that is made of six-to-ten word characters can be written like this: \A\w{6,10}\z  
The [\A anchor](https://www.rexegg.com/regex-anchors.html#A) asserts that the current position is the beginning of the string. After matching the six to ten word characters, the [\z anchor](https://www.rexegg.com/regex-anchors.html#z) asserts that the current position is the end of the string.  
  
Within a lookahead, this pattern becomes (?=\A\w{6,10}\z). This lookahead asserts: at the current position in the string, what follows is the beginning of the string, six to ten word characters, and the very end of the string.  
  
We want to make this assertion at the very beginning of the string. Therefore, to continue building our pattern, we want to anchor the lookahead with an \A. There is no need to duplicate the \A, so we can take it out of the lookahead. Our pattern becomes:  
\A(?=\w{6,10}\z)  
So far, we have an expression that validates that a string is entirely composed of six to ten word characters. Note that we haven't matched any of these characters yet: we have only looked ahead. The current position after the lookahead is still the beginning of the string. To check the other conditions, we just add lookaheads.  
  
Condition 2  
For our second condition, we need to check that the password contains one lowercase letter. To find one lowercase letter, the simplest idea is to use .\*[a-z]. That works, but the dot-star first shoots down to the end of the string, so we will always need to backtrack. Just for the sport, can we think of something more efficient? You might think of making the star quantifier reluctant by adding a ?, giving us .\*?[a-z], but that too requires backtracking as a [lazy quantifier requires backtracking at each step](https://www.rexegg.com/regex-quantifiers.html#lazy_expensive).  
  
For this type of situation, I recommend you use something like [^a-z]\*[a-z] (or even better, depending on your engine, the [atomic](https://www.rexegg.com/regex-disambiguation.html#atomic) (?>[^a-z]\*)[a-z] or [possessive](https://www.rexegg.com/regex-quantifiers.html#possessive) version [^a-z]\*+[a-z]—but we'll discuss that in the [footnotes](https://www.rexegg.com/regex-lookarounds.html#atomictweak)). The negated character class [^a-z] is the counterclass of the lowercase letter [a-z] we are looking for: it matches one character that is not a lowercase letter, and the \* quantifier makes us match zero or more such characters. The pattern [^a-z]\*[a-z] is a good example of the principle of [contrast](https://www.rexegg.com/regex-style.html#contrast) recommended by the regex style guide.  
  
Let's use this pattern inside a lookahead: (?=[^a-z]\*[a-z])  
The lookahead asserts: at this position in the string (i.e., the beginning of the string), we can match zero or more characters that are not lowercase letters, then we can match one lowercase letter: [a-z]  
Our pattern becomes:  
\A(?=\w{6,10}\z)(?=[^a-z]\*[a-z])  
At this stage, we have asserted that we are at the beginning of the string, and we have looked ahead twice. We still haven't matched any characters. Note that on a logical level it doesn't matter which condition we check first. If we swapped the order of the lookaheads, the result would be the same.  
  
We have two more conditions to satisfy: two more lookaheads.  
  
Condition 3  
For our third condition, we need to check that the password contains at least three uppercase letters. The logic is similar to condition 2: we look for an optional number of non-uppercase letters, then one uppercase letter… But we need to repeat that three times, for which we'll use the quantifier {3}.  
We'll use this lookahead: (?=(?:[^A-Z]\*[A-Z]){3})  
  
The lookahead asserts: at this position in the string (i.e., the beginning of the string), we can do the following three times: match zero or more characters that are not uppercase letters (the job of the negated character class [^A-Z] with the quantifier \*), then match one uppercase letter: [A-Z]  
Our pattern becomes:  
\A(?=\w{6,10}\z)(?=[^a-z]\*[a-z])(?=(?:[^A-Z]\*[A-Z]){3})  
At this stage, we have asserted that we are at the beginning of the string, and we have looked ahead three times. We still haven't matched any characters.  
  
Condition 4  
To check that the string contains at least one digit, we use this lookahead: (?=\D\*\d). Opposing \d to its counterclass \D makes good use of the [regex principle of contrast](https://www.rexegg.com/regex-style.html#contrast).  
  
The lookahead asserts: at this position in the string (i.e., the beginning of the string), we can match zero or more characters that are not digits (the job of the "not-a-digit" character class \D and the \* quantifier), then we can match one digit: \d  
Our pattern becomes:  
\A(?=\w{6,10}\z)(?=[^a-z]\*[a-z])(?=(?:[^A-Z]\*[A-Z]){3})(?=\D\*\d)  
At this stage, we have asserted that we are at the beginning of the string, and we have looked ahead four times to check our four conditions. We still haven't matched any characters, but we have validated our string: we know that it is a valid password.  
  
If all we wanted was to validate the password, we could stop right there. But if for any reason we also need to match and return the entire string—perhaps because we ran the regex on the output of a function and the password's characters haven't yet been assigned to a variable—we can easily do so now.  
  
Matching the Validated String  
After checking that the string conforms to all four conditions, we are still standing at the beginning of the string. The five assertions we have made (the anchor \A and the four lookaheads) have not changed our position. At this stage, we can use a simple .\* to gobble up the string: we know that whatever characters are matched by the dot-star, the string is a valid password. The pattern becomes:  
\A(?=\w{6,10}\z)(?=[^a-z]\*[a-z])(?=(?:[^A-Z]\*[A-Z]){3})(?=\D\*\d).\*  
[(direct link)](https://www.rexegg.com/regex-lookarounds.html#n-1conds)  
Fine-Tuning: Removing One Condition

For n conditions,  
use n-1 lookaheads

If you examine our lookaheads, you may notice that the pattern \w{6,10}\z inside the first one examines all the characters in the string. Therefore, we could have used this pattern to match the whole string instead of the dot-star .\*  
  
This allows us to remove one lookahead and to simplify the pattern to this:  
  
\A(?=[^a-z]\*[a-z])(?=(?:[^A-Z]\*[A-Z]){3})(?=\D\*\d)\w{6,10}\z  
The pattern \w{6,10}\z now serves the double purpose of matching the whole string and of ensuring that the string is entirely composed of six to ten word characters.  
  
Generalizing this result, if you must check for n conditions, your pattern only needs to include n-1 lookaheads at the most. Often, you are even able to combine several conditions into a single lookahead.  
  
You may object that we were able to use \w{6,10}\z because it happened to match the whole string. Indeed that was the case. But we could also have converted any of the other three lookaheads to match the entire string. For instance, taking the lookahead (?=\D\*\d) which checks for the presence of one digit, we can add a simple .\*\z to get us to the end of the string.  
  
The pattern would have become:  
\A(?=\w{6,10}\z)(?=[^a-z]\*[a-z])(?=(?:[^A-Z]\*[A-Z]){3})\D\*\d.\*\z  
By the way, you may wonder why I bother using the \z after the .\*: shouldn't it get me to the end of the string? In general, not so: unless we're in [DOTALL mode](https://www.rexegg.com/regex-modifiers.html#dotall), the dot doesn't match line breaks. Therefore, the .\* only gets you to the end of the first line. After this, the string may have line breaks and many more line. A \z anchor ensures that after the .\* we have reached not only the end of the line, but also the end of the string.  
  
In this particular pattern, the first lookaround (?=\w{6,10}\z) already ensures that there cannot be any line breaks in the string, so the final \z is not strictly necessary.

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

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.

Lookahead and Lookbehind Zero-Length Assertions

Lookahead and lookbehind, collectively called “lookaround”, are zero-length assertions just like the [start and end of line](https://www.regular-expressions.info/anchors.html), and [start and end of word](https://www.regular-expressions.info/wordboundaries.html) anchors explained earlier in this tutorial. The difference is that lookaround actually matches characters, but then gives up the match, returning only the result: match or no match. That is why they are called “assertions”. They do not consume characters in the string, but only assert whether a match is possible or not. Lookaround allows you to create regular expressions that are impossible to create without them, or that would get very longwinded without them.

Positive and Negative Lookahead

Negative lookahead is indispensable if you want to match something not followed by something else. When explaining [character classes](https://www.regular-expressions.info/charclass.html), this tutorial explained why you cannot use a negated character class to match a q not followed by a u. Negative lookahead provides the solution: q(?!u). 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. The positive lookahead construct is a pair of parentheses, with the opening parenthesis followed by a question mark and an equals sign.

You can use any regular expression inside the lookahead (but not lookbehind, as explained below). Any valid regular expression can be used inside the lookahead. If it contains [capturing groups](https://www.regular-expressions.info/brackets.html) then those groups will capture as normal and backreferences to them will work normally, even outside the lookahead. (The only exception is [Tcl](https://www.regular-expressions.info/tcl.html" \t "_top), which treats all groups inside lookahead as non-capturing.) The lookahead itself is not a capturing group. It is not included in the count towards numbering the backreferences. If you want to store the match of the regex inside a lookahead, you have to put capturing parentheses around the regex inside the lookahead, like this: (?=(regex)). The other way around will not work, because the lookahead will already have discarded the regex match by the time the capturing group is to store its match.

Regex Engine Internals

First, let’s see how the engine applies q(?!u) to the string Iraq. The first token in the regex is the [literal](https://www.regular-expressions.info/characters.html) q. As we already know, this causes the engine to traverse the string until the q in the string is matched. The position in the string is now the void after the string. The next token is the lookahead. The engine takes note that it is inside a lookahead construct now, and begins matching the regex inside the lookahead. So the next token is u. This does not match the void after the string. The engine notes that the regex inside the lookahead failed. Because the lookahead is negative, this means that the lookahead has successfully matched at the current position. At this point, the entire regex has matched, and q is returned as the match.

Let’s try applying the same regex to quit. q matches q. The next token is the u inside the lookahead. The next character is the u. These match. The engine advances to the next character: i. However, it is done with the regex inside the lookahead. The engine notes success, and discards the regex match. This causes the engine to step back in the string to u.

Because the lookahead is negative, the successful match inside it causes the lookahead to fail. Since there are no other permutations of this regex, the engine has to start again at the beginning. Since q cannot match anywhere else, the engine reports failure.

Let’s take one more look inside, to make sure you understand the implications of the lookahead. Let’s apply q(?=u)i to quit. The lookahead is now positive and is followed by another token. Again, q matches q and u matches u. Again, the match from the lookahead must be discarded, so the engine steps back from i in the string to u. The lookahead was successful, so the engine continues with i. But i cannot match u. So this match attempt fails. All remaining attempts fail as well, because there are no more q’s in the string.

The regex q(?=u)i can never match anything. It tries to match u and i at the same position. If there is a u immediately after the q then the lookahead succeeds but then i fails to match u. If there is anything other than a u immediately after the q then the lookahead fails.

Positive and Negative Lookbehind

Lookbehind has the same effect, but works backwards. It tells the regex engine to temporarily step backwards in the string, to check if the text inside the lookbehind can be matched there. (?<!a)b matches a “b” that is not preceded by an “a”, using negative lookbehind. It doesn’t match cab, but matches the b (and only the b) in bed or debt. (?<=a)b (positive lookbehind) matches the b (and only the b) in cab, but does not match bed or debt.

The construct for positive lookbehind is (?<=text): a pair of parentheses, with the opening parenthesis followed by a question mark, “less than” symbol, and an equals sign. Negative lookbehind is written as (?<!text), using an exclamation point instead of an equals sign.

More Regex Engine Internals

Let’s apply (?<=a)b to thingamabob. 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. So the lookbehind fails, and the engine starts again at the next character, the h. (Note that a negative lookbehind would have succeeded here.) Again, the engine temporarily steps back one character to check if an “a” can be found there. It finds a t, so the positive lookbehind fails again.

The lookbehind continues to fail until the regex reaches the m in the string. The engine again steps back one character, and notices that the a can be matched there. The positive lookbehind matches. Because it is zero-length, the current position in the string remains at the m. The next token is b, which cannot match here. The next character is the second a in the string. The engine steps back, and finds out that the m does not match a.

The next character is the first b in the string. The engine steps back and finds out that a satisfies the lookbehind. b matches b, and the entire regex has been matched successfully. It matches one character: the first b in the string.

Important Notes About Lookbehind

The good news is that you can use lookbehind anywhere in the regex, not only at the start. If you want to find a word not ending with an “s”, you could use \b\w+(?<!s)\b. This is definitely not the same as \b\w+[^s]\b. When applied to John's, the former matches John and the latter matches John' (including the apostrophe). I will leave it up to you to figure out why. (Hint: \b matches between the apostrophe and the s). The latter also doesn’t match single-letter words like “a” or “I”. The correct regex without using lookbehind is \b\w\*[^s\W]\b (star instead of plus, and \W in the character class). Personally, I find the lookbehind easier to understand. The last regex, which works correctly, has a double negation (the \W in the negated character class). Double negations tend to be confusing to humans. Not to regex engines, though. (Except perhaps for Tcl, which treats negated shorthands in negated character classes as an error.)

The bad news is that most regex flavors do not allow you to use just any regex inside a lookbehind, because they cannot apply a regular expression backwards. The regular expression engine needs to be able to figure out how many characters to step back before checking the lookbehind. When evaluating the lookbehind, the regex engine determines the length of the regex inside the lookbehind, steps back that many characters in the subject string, and then applies the regex inside the lookbehind from left to right just as it would with a normal regex.

Many regex flavors, including those used by [Perl](https://www.regular-expressions.info/perl.html), [Python](https://www.regular-expressions.info/python.html), and [Boost](https://www.regular-expressions.info/boost.html) only allow fixed-length strings. You can use [literal text](https://www.regular-expressions.info/characters.html), [character escapes](https://www.regular-expressions.info/nonprint.html#hex), [Unicode escapes](https://www.regular-expressions.info/nonprint.html#hex) other than \X, and [character classes](https://www.regular-expressions.info/charclass.html). You cannot use [quantifiers](https://www.regular-expressions.info/repeat.html) or [backreferences](https://www.regular-expressions.info/backref.html). You can use [alternation](https://www.regular-expressions.info/alternation.html), but only if all alternatives have the same length. These flavors evaluate lookbehind by first stepping back through the subject string for as many characters as the lookbehind needs, and then attempting the regex inside the lookbehind from left to right.

Perl 5.30 supports variable-length lookbehind as an experimental feature. But there are many cases in which it does not work correctly. So in practice, the above is still true for Perl 5.30.

[PCRE](https://www.regular-expressions.info/pcre.html) is not fully Perl-compatible when it comes to lookbehind. While Perl requires alternatives inside lookbehind to have the same length, PCRE allows alternatives of variable length. [PHP](https://www.regular-expressions.info/php.html), [Delphi](https://www.regular-expressions.info/delphi.html), [R](https://www.regular-expressions.info/rlanguage.html), and [Ruby](https://www.regular-expressions.info/ruby.html) also allow this. Each alternative still has to be fixed-length. Each alternative is treated as a separate fixed-length lookbehind.

[Java](https://www.regular-expressions.info/java.html) takes things a step further by allowing finite repetition. You can use the [question mark](https://www.regular-expressions.info/optional.html) and the [curly braces](https://www.regular-expressions.info/repeat.html) with the max parameter specified. Java determines the minimum and maximum possible lengths of the lookbehind. The lookbehind in the regex (?<!ab{2,4}c{3,5}d)test has 5 possible lengths. It can be from 7 through 11 characters long. When Java (version 6 or later) tries to match the lookbehind, it first steps back the minimum number of characters (7 in this example) in the string and then evaluates the regex inside the lookbehind as usual, from left to right. If it fails, Java steps back one more character and tries again. If the lookbehind continues to fail, Java continues to step back until the lookbehind either matches or it has stepped back the maximum number of characters (11 in this example). This repeated stepping back through the subject string kills performance when the number of possible lengths of the lookbehind grows. Keep this in mind. Don’t choose an arbitrarily large maximum number of repetitions to work around the lack of infinite quantifiers inside lookbehind. Java 4 and 5 have bugs that cause lookbehind with alternation or variable quantifiers to fail when it should succeed in some situations. These bugs were fixed in Java 6.

Java 13 allows you to use the [star](https://www.regular-expressions.info/repeat.html) and [plus](https://www.regular-expressions.info/repeat.html) inside lookbehind, as well as [curly braces](https://www.regular-expressions.info/repeat.html) without an upper limit. But Java 13 still uses the laborious method of matching lookbehind introduced with Java 6. Java 13 also does not correctly handle lookbehind with multiple quantifiers if one of them is unbounded. In some situations you may get an error. In other situations you may get incorrect matches. So for both correctness and performance, we recommend you only use quantifiers with a low upper bound in lookbehind with Java 6 through 13.

The only regex engines that allow you to use a full regular expression inside lookbehind, including infinite repetition and backreferences, are the [JGsoft engine](https://www.regular-expressions.info/jgsoft.html" \t "_top) and the [.NET RegEx classes](https://www.regular-expressions.info/dotnet.html). These regex engines really apply the regex inside the lookbehind backwards, going through the regex inside the lookbehind and through the subject string from right to left. They only need to evaluate the lookbehind once, regardless of how many different possible lengths it has.

Finally, flavors like [std::regex](https://www.regular-expressions.info/stdregex.html) and [Tcl](https://www.regular-expressions.info/tcl.html" \t "_top) do not support lookbehind at all, even though they do support lookahead. [JavaScript](https://www.regular-expressions.info/javascript.html) was like that for the longest time since its inception. But now lookbehind is part of the ECMAScript 2018 specification. As of this writing (late 2019), Google’s Chrome browser is the only popular JavaScript implementation that supports lookbehind. So if cross-browser compatibility matters, you can’t use lookbehind in JavaScript.

Lookaround Is Atomic

The fact that lookaround is zero-length automatically makes it [atomic](https://www.regular-expressions.info/atomic.html#use). As soon as the lookaround condition is satisfied, the regex engine forgets about everything inside the lookaround. It will not backtrack inside the lookaround to try different permutations.

The only situation in which this makes any difference is when you use [capturing groups](https://www.regular-expressions.info/brackets.html) inside the lookaround. Since the regex engine does not backtrack into the lookaround, it will not try different permutations of the capturing groups.

For this reason, the regex (?=(\d+))\w+\1 never matches 123x12. First the lookaround captures 123 into \1. \w+ then matches the whole string and backtracks until it matches only 1. Finally, \w+ fails since \1 cannot be matched at any position. Now, the regex engine has nothing to backtrack to, and the overall regex fails. The backtracking steps created by \d+ have been discarded. It never gets to the point where the lookahead captures only 12.

Obviously, the regex engine does try further positions in the string. If we change the subject string, the regex (?=(\d+))\w+\1 does match 56x56 in 456x56.

If you don’t use capturing groups inside lookaround, then all this doesn’t matter. Either the lookaround condition can be satisfied or it cannot be. In how many ways it can be satisfied is irrelevant

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

In this article, we show how to use named groups with regular expressions in Python.

Groups are used in Python in order to reference regular expression matches.

By default, groups, without names, are referenced according to numerical order starting with 1 .

Let's say we have a regular expression that has 3 subexpressions.

A user enters in his birthdate, according to the month, day, and year.

Let's say the user must first enter the month, then the day, and then the year.

Using the group() function in Python, without named groups, the first match (the month) would be referenced using the statement, group(1). The second match (the day) would be referenced using the statement, group(2). The third match (the year) would be referenced using the statment, group(3).

Now, with named groups, we can name each match in the regular expression. So instead of referencing matches of the regular expression with numbers (group(1), group(2), etc.), we can reference matches with names, such as group('month'), group('day'), group('year').

Named groups makes the code more organized and more readable.

By seeing, group(1), you don't really know what this represents.

But if you see, group('month') or group('year'), you know it's referencing the month or the year.

So named groups makes code more readable and more understandable rather than the default numerical referencing.

So let's go over some code and see an actual real-world example of named groups in Python.

o let's now go over this code.

re is the module in Python that allows us to use regular expressions. So we first have to import re in our code, in order to use regular expressions.

After this, we have a variable, string1, which is set equal to a date, June 15, 1987.

We then have a variable, regex, which is set equal to, r"^(?P\w+)\s(?P\d+)\,?\s(?P\d+)"

Let's break this regular expression down now.

So when we want to create a named group, the expression to do so is, (?Pcontent), where the name of the named group is namedgroup and it content is where you see content.

In our regular expression, the first named group is the month and this consists of 1 or more alphabetical characters.

A space then ensues.

The second named group is day. This consists of 1 or more digits.

This is followed by an optional character and a space.

The third named group is year. This consists of 1 or more digits.

We then look up matches with the statement, matches= re.search(regex, string1)

The matches get stored in the variable, matches

We then can output the month by the statement, matches.group('month')

We can output the day by the statement, matches.group('day')

We can output the year by the statement, matches.group('year')

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.

And this is how we can use named groups with regular expressions in Python

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

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);

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

 $&

Refers to the entire matched string.

 $n

Refers to the nth group within the matched string. For example, “$1” refers to the first group of characters tagged by the regex object; “$2” refers to the second group of tagged characters (if there is one), and so on. The example that follows should clarify.

 $$

A literal dollar sign ($).

The following declarations set up a search-and-replace designed to fix the repeated-word pattern, replacing it, where found, with one copy of the word.

using std::regex;

using std::regex\_replace;

using std::string;

regex reg1("([A-Za-z]+) \\1"); // Find double word.

string replacement = "$1"; // Replace with one word.

With these objects defined, the following statements execute search-and-replace on the string shown earlier.

string target = "The cow cow jumped over the the moon.";

string result = regex\_replace(target, reg1, replacement);

std::cout << result << std::endl;

The output is:

The cow jumped over the moon.

which is what we wanted.

Let’s review how this works. When the text “cow cow” was matched by the regular-expression object, the first occurrence of “cow” was tagged because it matched the expression inside the parentheses: “([A-Za-z]+)”. The rest of the expression, “\\1”, indicated that the regex object then needed to match a space, followed by a recurrence of the tagged characters, to match the overall expression. Therefore, “cow” gets tagged and “cow cow” matches the entire regular expression.

The replacement pattern, “$1”, causes the matched text—“cow cow”—to be replaced by “cow”, the tagged group. Suppose the replacement pattern were “XX$1YY$1ZZ$1”. Then the replacement text would have been “XXcowYYcowZZcow” and that would have replaced “cow cow”.

Characters not matched by the regex object, reg1, are just copied into the result as they are. So, for example, the words “jumped over” are copied without being transformed.

Here’s an example of another regex object and replacement-pattern string: When used with the call to regex\_replace shown earlier, these result in the switching of two words separated by an ampersand (&). For example, “boy&girl” would be replaced by “girl&boy” and vice versa.

regex reg1("([A-Za-z]+)&([A-Za-z]+)"); // Find word&word

string replacement = "$2&$1"; // Switch order.

The regex\_replace function is particularly convenient. It isn’t necessary to iterate through the target string. Instead, regex\_replace carries out replacements on all the substrings matching the pattern in the regex object, while leaving the rest of the text alone.

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?

What is Regular Expression in Python?

A Regular Expression (RE) in a programming language is a special text string used for describing a search pattern. It is extremely useful for extracting information from text such as code, files, log, spreadsheets or even documents.

While using the Python regular expression the first thing is to recognize is that everything is essentially a character, and we are writing patterns to match a specific sequence of characters also referred as string. Ascii or latin letters are those that are on your keyboards and Unicode is used to match the foreign text. It includes digits and punctuation and all special characters like $#@!%, etc.

In this Python RegEx tutorial, we will learn-

[Regular Expression Syntax](https://www.guru99.com/python-regular-expressions-complete-tutorial.html#1)

[Example of w+ and ^ Expression](https://www.guru99.com/python-regular-expressions-complete-tutorial.html#2)

[Example of \s expression in re.split function](https://www.guru99.com/python-regular-expressions-complete-tutorial.html#3)

[Using regular expression methods](https://www.guru99.com/python-regular-expressions-complete-tutorial.html#4)

[Using re.match()](https://www.guru99.com/python-regular-expressions-complete-tutorial.html#5)

[Finding Pattern in Text (re.search())](https://www.guru99.com/python-regular-expressions-complete-tutorial.html#6)

[Using re.findall for text](https://www.guru99.com/python-regular-expressions-complete-tutorial.html#7)

[Python Flags](https://www.guru99.com/python-regular-expressions-complete-tutorial.html#8)

[Example of re.M or Multiline Flags](https://www.guru99.com/python-regular-expressions-complete-tutorial.html#9)

For instance, a Python regular expression could tell a program to search for specific text from the string and then to print out the result accordingly. Expression can include

Text matching

Repetition

Branching

Pattern-composition etc.

Regular expression or RegEx in Python is denoted as RE (REs, regexes or regex pattern) are imported through re module. Python supports regular expression through libraries. RegEx in Python supports various things like Modifiers, Identifiers, and White space characters.

| Identifiers | Modifiers | White space characters | Escape required |
| --- | --- | --- | --- |
| \d= any number (a digit) | \d represents a digit.Ex: \d{1,5} it will declare digit between 1,5 like 424,444,545 etc. | \n = new line | . + \* ? [] $ ^ () {} | \ |
| \D= anything but a number (a non-digit) | + = matches 1 or more | \s= space |  |
| \s = space (tab,space,newline etc.) | ? = matches 0 or 1 | \t =tab |  |
| \S= anything but a space | \* = 0 or more | \e = escape |  |
| \w = letters ( Match alphanumeric character, including “\_”) | $ match end of a string | \r = carriage return |  |
| \W =anything but letters ( Matches a non-alphanumeric character excluding “\_”) | ^ match start of a string | \f= form feed |  |
| . = anything but letters (periods) | | matches either or x/y | —————– |  |
| \b = any character except for new line | [] = range or “variance” | —————- |  |
| \. | {x} = this amount of preceding code | —————– |  |

Regular Expression(RE) Syntax

import re

“re” module included with Python primarily used for string searching and manipulation

Also used frequently for web page “Scraping” (extract large amount of data from websites)

We will begin the expression tutorial with this simple exercise by using the expressions (w+) and (^).

Example of w+ and ^ Expression

“^”: This expression matches the start of a string

“w+“: This expression matches the alphanumeric character in the string

Here we will see a Python RegEx Example of how we can use w+ and ^ expression in our code. We cover the function re.findall() in Python, later in this tutorial but for a while we simply focus on \w+ and \^ expression.

For example, for our string “guru99, education is fun” if we execute the code with w+ and^, it will give the output “guru99”.

import re

xx = "guru99,education is fun"

r1 = re.findall(r"^\w+",xx)

print(r1)

Remember, if you remove +sign from the w+, the output will change, and it will only give the first character of the first letter, i.e., [g]

Example of \s expression in re.split function

“s”: This expression is used for creating a space in the string

To understand how this RegEx in Python works, we begin with a simple Python RegEx Example of a split function. In the example, we have split each word using the “re.split” function and at the same time we have used expression \s that allows to parse each word in the string separately.

When you execute this code it will give you the output [‘we’, ‘are’, ‘splitting’, ‘the’, ‘words’].

Now, let see what happens if you remove “\” from s. There is no ‘s’ alphabet in the output, this is because we have removed ‘\’ from the string, and it evaluates “s” as a regular character and thus split the words wherever it finds “s” in the string.

Similarly, there are series of other Python regular expression that you can use in various ways in Python like \d,\D,$,\.,\b, etc.

Here is the complete code

import re

xx = "guru99,education is fun"

r1 = re.findall(r"^\w+", xx)

print((re.split(r'\s','we are splitting the words')))

print((re.split(r's','split the words')))

Next, we will going to see the types of methods that are used with regular expression in Python.

Using regular expression methods

The “re” package provides several methods to actually perform queries on an input string. We will see the methods of re in Python:

re.match()

re.search()

re.findall()

Note: Based on the regular expressions, Python offers two different primitive operations. The match method checks for a match only at the beginning of the string while search checks for a match anywhere in the string.

re.match()

re.match() function of re in Python will search the regular expression pattern and return the first occurrence. The Python RegEx Match method checks for a match only at the beginning of the string. So, if a match is found in the first line, it returns the match object. But if a match is found in some other line, the Python RegEx Match function returns null.

For example, consider the following code of Python re.match() function. The expression “w+” and “\W” will match the words starting with letter ‘g’ and thereafter, anything which is not started with ‘g’ is not identified. To check match for each element in the list or string, we run the forloop in this Python re.match() Example.

re.search(): Finding Pattern in Text

re.search() function will search the regular expression pattern and return the first occurrence. Unlike Python re.match(), it will check all lines of the input string. The Python re.search() function returns a match object when the pattern is found and “null” if the pattern is not found

How to use search()?

In order to use search() function, you need to import Python re module first and then execute the code. The Python re.search() function takes the “pattern” and “text” to scan from our main string

For example here we look for two literal strings “Software testing” “guru99”, in a text string “Software[Testing](https://www.guru99.com/software-testing.html)is fun”. For “software testing” we found the match hence it returns the output of Python re.search() Example as “found a match”, while for word “guru99” we could not found in string hence it returns the output as “No match”.

re.findall()

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

How to Use re.findall() in Python?

Here we have a list of e-mail addresses, and we want all the e-mail addresses to be fetched out from the list, we use the method re.findall() in Python. It will find all the e-mail addresses from the list.

Here is the complete code for Example of re.findall()

import re

list = ["guru99 get", "guru99 give", "guru Selenium"]

for element in list:

z = re.match("(g\w+)\W(g\w+)", element)

if z:

print((z.groups()))

patterns = ['software testing', 'guru99']

text = 'software testing is fun?'

for pattern in patterns:

print('Looking for "%s" in "%s" ->' % (pattern, text), end=' ')

if re.search(pattern, text):

print('found a match!')

else:

print('no match')

abc = 'guru99@google.com, careerguru99@hotmail.com, users@yahoomail.com'

emails = re.findall(r'[\w\.-]+@[\w\.-]+', abc)

for email in emails:

print(email

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

Class Scanner

[java.lang.Object](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)

java.util.Scanner

All Implemented Interfaces:

[Closeable](https://docs.oracle.com/javase/7/docs/api/java/io/Closeable.html), [AutoCloseable](https://docs.oracle.com/javase/7/docs/api/java/lang/AutoCloseable.html" \o "interface in java.lang), [Iterator](https://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html)<[String](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html)>

public final class Scanner

extends [Object](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)

implements [Iterator](https://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html)<[String](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html)>, [Closeable](https://docs.oracle.com/javase/7/docs/api/java/io/Closeable.html)

A simple text scanner which can parse primitive types and strings using regular expressions.

A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods.

For example, this code allows a user to read a number from System.in:

Scanner sc = new Scanner(System.in);

int i = sc.nextInt();

As another example, this code allows long types to be assigned from entries in a file myNumbers:

Scanner sc = new Scanner(new File("myNumbers"));

while (sc.hasNextLong()) {

long aLong = sc.nextLong();

}

The scanner can also use delimiters other than whitespace. This example reads several items in from a string:

String input = "1 fish 2 fish red fish blue fish";

Scanner s = new Scanner(input).useDelimiter("\\s\*fish\\s\*");

System.out.println(s.nextInt());

System.out.println(s.nextInt());

System.out.println(s.next());

System.out.println(s.next());

s.close();

prints the following output:

1

2

red

blue

The same output can be generated with this code, which uses a regular expression to parse all four tokens at once:

String input = "1 fish 2 fish red fish blue fish";

Scanner s = new Scanner(input);

s.findInLine("(\\d+) fish (\\d+) fish (\\w+) fish (\\w+)");

MatchResult result = s.match();

for (int i=1; i<=result.groupCount(); i++)

System.out.println(result.group(i));

s.close();

The default whitespace delimiter used by a scanner is as recognized by [Character](https://docs.oracle.com/javase/7/docs/api/java/lang/Character.html" \o "class in java.lang).[isWhitespace](https://docs.oracle.com/javase/7/docs/api/java/lang/Character.html#isWhitespace(char)). The [reset()](https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html#reset()) method will reset the value of the scanner's delimiter to the default whitespace delimiter regardless of whether it was previously changed.

A scanning operation may block waiting for input.

The [next()](https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html#next()) and [hasNext()](https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html" \l "hasNext()) methods and their primitive-type companion methods (such as [nextInt()](https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html" \l "nextInt()) and [hasNextInt()](https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html" \l "hasNextInt())) first skip any input that matches the delimiter pattern, and then attempt to return the next token. Both hasNext and next methods may block waiting for further input. Whether a hasNext method blocks has no connection to whether or not its associated next method will block.

The [findInLine(java.lang.String)](https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html#findInLine(java.lang.String)), [findWithinHorizon(java.lang.String, int)](https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html#findWithinHorizon(java.lang.String,%20int)), and [skip(java.util.regex.Pattern)](https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html#skip(java.util.regex.Pattern)) methods operate independently of the delimiter pattern. These methods will attempt to match the specified pattern with no regard to delimiters in the input and thus can be used in special circumstances where delimiters are not relevant. These methods may block waiting for more input.

When a scanner throws an [InputMismatchException](https://docs.oracle.com/javase/7/docs/api/java/util/InputMismatchException.html" \o "class in java.util), the scanner will not pass the token that caused the exception, so that it may be retrieved or skipped via some other method.

Depending upon the type of delimiting pattern, empty tokens may be returned. For example, the pattern "\\s+" will return no empty tokens since it matches multiple instances of the delimiter. The delimiting pattern "\\s" could return empty tokens since it only passes one space at a time.

A scanner can read text from any object which implements the [Readable](https://docs.oracle.com/javase/7/docs/api/java/lang/Readable.html) interface. If an invocation of the underlying readable's [Readable.read(java.nio.CharBuffer)](https://docs.oracle.com/javase/7/docs/api/java/lang/Readable.html" \l "read(java.nio.CharBuffer)) method throws an [IOException](https://docs.oracle.com/javase/7/docs/api/java/io/IOException.html" \o "class in java.io) then the scanner assumes that the end of the input has been reached. The most recent IOException thrown by the underlying readable can be retrieved via the [ioException()](https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html" \l "ioException()) method.

When a Scanner is closed, it will close its input source if the source implements the [Closeable](https://docs.oracle.com/javase/7/docs/api/java/io/Closeable.html) interface.

A Scanner is not safe for multithreaded use without external synchronization.

Unless otherwise mentioned, passing a null parameter into any method of a Scanner will cause a NullPointerException to be thrown.

A scanner will default to interpreting numbers as decimal unless a different radix has been set by using the [useRadix(int)](https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html" \l "useRadix(int)) method. The [reset()](https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html#reset()) method will reset the value of the scanner's radix to 10 regardless of whether it was previously changed.