Q1. What is the benefit of regular expressions?

Regular expressions are useful in search and replace operations. The typical use case is to look for a sub-string that matches a pattern and replace it with something else. Most APIs using regular expressions allow you to reference capture groups from the search pattern in the replacement string

Regular expressions are useful in any scenario that benefits from full or partial pattern matches on strings. These are some common use cases:

verify the structure of strings

extract substrings form structured strings

search / replace / rearrange parts of the string

split a string into tokens

All of these come up regularly when doing data preparation work.

Q2. How much do you need to use the following sentence while using regular expressions?

The following examples illustrate the use and construction of simple regular expressions. Each example includes the type of text to match, one or more regular expressions that match that text, and notes that explain the use of the special characters and formatting.

[Match Exact Phrase Only](https://support.google.com/a/answer/1371417?hl=en#Match-Exact-Phrase-Only)

[Match Word or Phrase in a List](https://support.google.com/a/answer/1371417?hl=en#Match-Word-or-Phrase-in-a-List)

[Match Word with Different Spellings or Special Characters](https://support.google.com/a/answer/1371417?hl=en#Match-Word-with-Different-Spellings)

[Match Any Email Address from a Specific Domain](https://support.google.com/a/answer/1371417?hl=en#Match-Any-Email-Address-from-a-Spec)

[Match Any IP Address in a Range](https://support.google.com/a/answer/1371417?hl=en#Match-Any-IP-Address-in-a-Range)

[Match an Alphanumeric Format](https://support.google.com/a/answer/1371417?hl=en#Match-an-Alphanumeric-Format)

For additional instructions and guidelines, see also [Guidelines for using regular expressions](https://support.google.com/a/answer/1346938) and [RE2 Syntax](http://code.google.com/p/re2/wiki/Syntax). See also [Set up rules for content compliance](https://support.google.com/a/answer/1346934).

Important: We support [RE2 Syntax](http://code.google.com/p/re2/wiki/Syntax) only, which differs slightly from PCRE. Regular expressions are case-sensitive by default.

Note: Examples shown below can be useful as starting points for more complex regular expressions. However, for matching a single word, we suggest that you use the Content compliance or Objectionable content settings.

|  |
| --- |
|  |
| Notes |

|  |  |
| --- | --- |
| Notes | \W matches any character that’s not a letter, digit, or underscore. It prevents the regex from matching characters before or after the email address.  ^ matches the start of a new line. Allows the regex to match the address if it appears at the beginning of a line, with no characters before it.  $ matches the end of a line. Allows the regex to match the address if it appears at the end of a line, with no characters after it.  [\w.\-] matches any word character (a-z, A-Z, 0-9, or an underscore), a period, or a hyphen. These are the most commonly used valid characters in the first part of an email address. The \- (which indicates a hyphen) must occur last in the list of characters within the square brackets.  The \ before the dash and period “escapes” these characters—that is, it indicates that the dash and period aren't a regex special characters themselves. There's no need to escape the period within the square brackets.  {0,25} indicates that from 0 to 25 characters in the preceding character set can occur before the @ symbol. The [Content Compliance email setting](https://support.google.com/a/answer/1346934) supports matching of up to 25 characters for each character set in a regular expression.  The (...) formatting groups the domains, and the | character that separates them indicates an “or.” |

|  |  |
| --- | --- |
|  |  |

Q3. Which characters have special significance in square brackets when expressing a range, and under what circumstances?

Sets and ranges [...]

Several characters or character classes inside square brackets […] mean to “search for any character among given”.

[Sets](https://javascript.info/regexp-character-sets-and-ranges" \l "sets)

For instance, [eao] means any of the 3 characters: 'a', 'e', or 'o'.

That’s called a set. Sets can be used in a regexp along with regular characters:

// find [t or m], and then "op"

alert( "Mop top".match(/[tm]op/gi) ); // "Mop", "top"

Please note that although there are multiple characters in the set, they correspond to exactly one character in the match.

So the example below gives no matches:

// find "V", then [o or i], then "la"

alert( "Voila".match(/V[oi]la/) ); // null, no matches

The pattern searches for:

V,

then one of the letters [oi],

then la.

So there would be a match for Vola or Vila.

[Ranges](https://javascript.info/regexp-character-sets-and-ranges" \l "ranges)

Square brackets may also contain character ranges.

For instance, [a-z] is a character in range from a to z, and [0-5] is a digit from 0 to 5.

In the example below we’re searching for "x" followed by two digits or letters from A to F:

alert( "Exception 0xAF".match(/x[0-9A-F][0-9A-F]/g) ); // xAF

Here [0-9A-F] has two ranges: it searches for a character that is either a digit from 0 to 9 or a letter from A to F.

If we’d like to look for lowercase letters as well, we can add the range a-f: [0-9A-Fa-f]. Or add the flag i.

We can also use character classes inside […].

For instance, if we’d like to look for a wordly character \w or a hyphen -, then the set is [\w-].

Combining multiple classes is also possible, e.g. [\s\d] means “a space character or a digit”.

Character classes are shorthands for certain character sets

For instance:

\d – is the same as [0-9],

\w – is the same as [a-zA-Z0-9\_],

\s – is the same as [\t\n\v\f\r ], plus few other rare Unicode space characters.

[Example: multi-language \w](https://javascript.info/regexp-character-sets-and-ranges" \l "example-multi-language-w)

As the character class \w is a shorthand for [a-zA-Z0-9\_], it can’t find Chinese hieroglyphs, Cyrillic letters, etc.

We can write a more universal pattern, that looks for wordly characters in any language. That’s easy with Unicode properties: [\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join\_C}].

Let’s decipher it. Similar to \w, we’re making a set of our own that includes characters with following Unicode properties:

Alphabetic (Alpha) – for letters,

Mark (M) – for accents,

Decimal\_Number (Nd) – for digits,

Connector\_Punctuation (Pc) – for the underscore '\_' and similar characters,

Join\_Control (Join\_C) – two special codes 200c and 200d, used in ligatures, e.g. in Arabic.

An example of use:

let regexp = /[\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join\_C}]/gu;

let str = `Hi 你好 12`;

// finds all letters and digits:

alert( str.match(regexp) );

Q4. How does compiling a regular-expression object benefit you?

compile(pattern, repl, string): We can combine a regular expression pattern into pattern objects, which can be used for pattern matching. It also helps to search a pattern again without rewriting it.

The re.compile() method

re.compile(pattern, repl, string):

 We can combine a regular expression pattern into pattern objects, which can be used for pattern matching. It also helps to search a pattern again without rewriting it.

Example

import re

pattern=re.compile('TP')

result=pattern.findall('TP Tutorialspoint TP')

print result

result2=pattern.findall('TP is most popular tutorials site of India')

print result2

Output

['TP', 'TP']

['TP']

Python’s re.compile() method is used to compile a regular expression pattern provided as a string into a regex pattern object (re.Pattern). Later we can use this pattern object to search for a match inside different target strings using regex methods such as a [re.match()](https://pynative.com/python-regex-pattern-matching/) or [re.search()](https://pynative.com/python-regex-search/).

In simple terms, We can compile a regular expression into a regex object to look for occurrences of the same pattern inside various target strings without rewriting it.

Q5. What are some examples of how to use the match object returned by re.match and re.search?

This module provides regular expression matching operations similar to those found in Perl.

Both patterns and strings to be searched can be Unicode strings ([str](https://docs.python.org/3/library/stdtypes.html#str)) as well as 8-bit strings ([bytes](https://docs.python.org/3/library/stdtypes.html#bytes)). However, Unicode strings and 8-bit strings cannot be mixed: that is, you cannot match a Unicode string with a byte pattern or vice-versa; similarly, when asking for a substitution, the replacement string must be of the same type as both the pattern and the search string.

Regular expressions use the backslash character ('\') to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python’s usage of the same character for the same purpose in string literals; for example, to match a literal backslash, one might have to write '\\\\' as the pattern string, because the regular expression must be \\, and each backslash must be expressed as \\ inside a regular Python string literal. Also, please note that any invalid escape sequences in Python’s usage of the backslash in string literals now generate a [DeprecationWarning](https://docs.python.org/3/library/exceptions.html" \l "DeprecationWarning" \o "DeprecationWarning) and in the future this will become a [SyntaxError](https://docs.python.org/3/library/exceptions.html" \l "SyntaxError" \o "SyntaxError). This behaviour will happen even if it is a valid escape sequence for a regular expression.

The solution is to use Python’s raw string notation for regular expression patterns; backslashes are not handled in any special way in a string literal prefixed with 'r'. So r"\n" is a two-character string containing '\' and 'n', while "\n" is a one-character string containing a newline. Usually patterns will be expressed in Python code using this raw string notation.

It is important to note that most regular expression operations are available as module-level functions and methods on [compiled regular expressions](https://docs.python.org/3/library/re.html#re-objects). The functions are shortcuts that don’t require you to compile a regex object first, but miss some fine-tuning parameters.

See also

The third-party [regex](https://pypi.org/project/regex/) module, which has an API compatible with the standard library [re](https://docs.python.org/3/library/re.html#module-re) module, but offers additional functionality and a more thorough Unicode support.

Regular Expression Syntax

A regular expression (or RE) specifies a set of strings that matches it; the functions in this module let you check if a particular string matches a given regular expression (or if a given regular expression matches a particular string, which comes down to the same thing).

Regular expressions can be concatenated to form new regular expressions; if A and B are both regular expressions, then AB is also a regular expression. In general, if a string p matches A and another string q matches B, the string pq will match AB. This holds unless A or B contain low precedence operations; boundary conditions between A and B; or have numbered group references. Thus, complex expressions can easily be constructed from simpler primitive expressions like the ones described here. For details of the theory and implementation of regular expressions, consult the Friedl book [[Frie09]](https://docs.python.org/3/library/re.html#frie09), or almost any textbook about compiler construction.

A brief explanation of the format of regular expressions follows. For further information and a gentler presentation, consult the [Regular Expression HOWTO](https://docs.python.org/3/howto/regex.html#regex-howto).

Regular expressions can contain both special and ordinary characters. Most ordinary characters, like 'A', 'a', or '0', are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so last matches the string 'last'. (In the rest of this section, we’ll write RE’s in this special style, usually without quotes, and strings to be matched 'in single quotes'.)

Some characters, like '|' or '(', are special. Special characters either stand for classes of ordinary characters, or affect how the regular expressions around them are interpreted.

Repetition operators or quantifiers (\*, +, ?, {m,n}, etc) cannot be directly nested. This avoids ambiguity with the non-greedy modifier suffix ?, and with other modifiers in other implementations. To apply a second repetition to an inner repetition, parentheses may be used. For example, the expression (?:a{6})\* matches any multiple of six 'a' characters.

The special characters are:

.

(Dot.) In the default mode, this matches any character except a newline. If the [DOTALL](https://docs.python.org/3/library/re.html#re.DOTALL) flag has been specified, this matches any character including a newline.

^

(Caret.) Matches the start of the string, and in [MULTILINE](https://docs.python.org/3/library/re.html#re.MULTILINE) mode also matches immediately after each newline.

$

Matches the end of the string or just before the newline at the end of the string, and in [MULTILINE](https://docs.python.org/3/library/re.html#re.MULTILINE) mode also matches before a newline. foo matches both ‘foo’ and ‘foobar’, while the regular expression foo$ matches only ‘foo’. More interestingly, searching for foo.$ in 'foo1\nfoo2\n' matches ‘foo2’ normally, but ‘foo1’ in [MULTILINE](https://docs.python.org/3/library/re.html#re.MULTILINE) mode; searching for a single $ in 'foo\n' will find two (empty) matches: one just before the newline, and one at the end of the string.

\*

Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. ab\* will match ‘a’, ‘ab’, or ‘a’ followed by any number of ‘b’s.

+

Causes the resulting RE to match 1 or more repetitions of the preceding RE. ab+ will match ‘a’ followed by any non-zero number of ‘b’s; it will not match just ‘a’.

?

Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. ab? will match either ‘a’ or ‘ab’.

\*?, +?, ??

The '\*', '+', and '?' quantifiers are all greedy; they match as much text as possible. Sometimes this behaviour isn’t desired; if the RE <.\*> is matched against '<a> b <c>', it will match the entire string, and not just '<a>'. Adding ? after the quantifier makes it perform the match in non-greedy or minimal fashion; as few characters as possible will be matched. Using the RE <.\*?> will match only '<a>'.

\*+, ++, ?+

Like the '\*', '+', and '?' quantifiers, those where '+' is appended also match as many times as possible. However, unlike the true greedy quantifiers, these do not allow back-tracking when the expression following it fails to match. These are known as possessive quantifiers. For example, a\*a will match 'aaaa' because the a\* will match all 4 'a's, but, when the final 'a' is encountered, the expression is backtracked so that in the end the a\* ends up matching 3 'a's total, and the fourth 'a' is matched by the final 'a'. However, when a\*+a is used to match 'aaaa', the a\*+ will match all 4 'a', but when the final 'a' fails to find any more characters to match, the expression cannot be backtracked and will thus fail to match. x\*+, x++ and x?+ are equivalent to (?>x\*), (?>x+) and (?>x?) correspondingly.

New in version 3.11.

{m}

Specifies that exactly m copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, a{6} will match exactly six 'a' characters, but not five.

{m,n}

Causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, a{3,5} will match from 3 to 5 'a' characters. Omitting m specifies a lower bound of zero, and omitting n specifies an infinite upper bound. As an example, a{4,}b will match 'aaaab' or a thousand 'a' characters followed by a 'b', but not 'aaab'. The comma may not be omitted or the modifier would be confused with the previously described form.

{m,n}?

Causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as few repetitions as possible. This is the non-greedy version of the previous quantifier. For example, on the 6-character string 'aaaaaa', a{3,5} will match 5 'a' characters, while a{3,5}? will only match 3 characters.

{m,n}+

Causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as many repetitions as possible without establishing any backtracking points. This is the possessive version of the quantifier above. For example, on the 6-character string 'aaaaaa', a{3,5}+aa attempt to match 5 'a' characters, then, requiring 2 more 'a's, will need more characters than available and thus fail, while a{3,5}aa will match with a{3,5} capturing 5, then 4 'a's by backtracking and then the final 2 'a's are matched by the final aa in the pattern. x{m,n}+ is equivalent to (?>x{m,n}).

Q6. What is the difference between using a vertical bar (|) as an alteration and using square brackets as a character set?

What is difference [] and () in regex?

[] denotes a character class. () denotes a capturing group. [a-z0-9] -- One character that is in the range of a-z OR 0-9. (a-z0-9) -- Explicit capture of a-z0-9 .

1

The first has characters in a character class, the second uses a capturing group. Those expressions are actually completely different. The first will match 0 or 1 whitespace character OR a +, followed by a comma, followed by 0 or 1 whitespace character OR a +. The latter will match 0 or more whitespace characters followed by a comma, followed by 0 or more whitespace characters

So, basically, if you have a string as follows: var myStr = ' , ', the first expression will NOT match, whereas the second one will. If you have another string as follows: var myOtherStr = '+,', it will match the first expression but not the second

I used JS as my example language, but the concept is how most regular expressions will work. [regular-expressions.info](http://www.regular-expressions.info/) is a great resource if you're trying to learn about regular expressions

A + is a special character in pretty much any flavor of regex. However, when inside of a character group (a set of characters inside of []), the + is matched as a regular character instead.

(…) is a [group](http://www.regular-expressions.info/brackets.html) that groups the contents like in math; (a-z0-9) is the grouped sequence of a-z0-9. Groups are particularly used with [quantifiers](http://www.regular-expressions.info/repeat.html) that allow the preceding expression to be repeated as a whole: a\*b\* matches any number of a’s followed by any number of b’s, e.g. a, aaab, bbbbb, etc.; in contrast to that, (ab)\* matches any number of ab’s, e.g. ab, abababab, etc

[…] is a [character class](http://www.regular-expressions.info/charclass.html) that describes the options for one single character; [a-z0-9] describes one single character that can be of the range a–z or 0–9

Q7. In regular-expression search patterns, why is it necessary to use the raw-string indicator (r)? In   replacement strings?

From the python documentation on [regex](https://docs.python.org/library/re.html), regarding the '\' character:

The solution is to use Python’s raw string notation for regular expression patterns; backslashes are not handled in any special way in a string literal prefixed with 'r'. So r"\n" is a two-character string containing '\' and 'n', while "\n" is a one-character string containing a newline. Usually patterns will be expressed in Python code using this raw string notation.

What is this raw string notation? If you use a raw string format, does that mean "\*" is taken as a a literal character rather than a zero-or-more indicator? That obviously can't be right, or else regex would completely lose its power. But then if it's a raw string, how does it recognize newline characters if "\n" is literally a backslash and an "n"?

I don't follow.

Edit for bounty:

I'm trying to understand how a raw string regex matches newlines, tabs, and character sets, e.g. \w for words or \d for digits or all whatnot, if raw string patterns don't recognize backslashes as anything more than ordinary characters. I could really use some good examples.

+50

You will perhaps find this easier to understand if you stop using the terms "raw string regex" and "raw string patterns". These terms conflate two separate concepts: the representations of a particular string in Python source code, and what regular expression that string represents.

In fact, it's helpful to think of these as two different programming languages, each with their own syntax. The Python language has source code that, among other things, builds strings with certain contents, and calls the regular expression system. The regular expression system has source code that resides in string objects, and matches strings. Both languages use backslash as an escape character.

First, understand that a string is a sequence of characters (i.e. bytes or Unicode code points; the distinction doesn't much matter here). There are many ways to represent a string in Python source code. A raw string is simply one of these representations. If two representations result in the same sequence of characters, they produce equivalent behaviour.

Imagine a 2-character string, consisting of the backslash character followed by the n character. If you know that the character value for backslash is 92, and for n is 110, then this expression generates our string:

s = chr(92)+chr(110)

print len(s), s

2 \n

The conventional Python string notation "\n" does not generate this string. Instead it generates a one-character string with a newline character. The [Python docs 2.4.1. String literals](http://docs.python.org/2/reference/lexical_analysis.html#string-literals) say, "The backslash (\) character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character."

s = "\n"

print len(s), s

1

(Note that the newline isn't visible in this example, but if you look carefully, you'll see a blank line after the "1".)

To get our two-character string, we have to use another backslash character to escape the special meaning of the original backslash character:

s = "\\n"

print len(s), s

2 \n

What if you want to represent strings that have many backslash characters in them? [Python docs 2.4.1. String literals](http://docs.python.org/2/reference/lexical_analysis.html#string-literals) continue, "String literals may optionally be prefixed with a letter 'r' or 'R'; such strings are called raw strings and use different rules for interpreting backslash escape sequences." Here is our two-character string, using raw string representation:

s = r"\n"

print len(s), s

2 \n

So we have three different string representations, all giving the same string, or sequence of characters:

print chr(92)+chr(110) == "\\n" == r"\n"

True

Now, let's turn to regular expressions. The [Python docs, 7.2. re — Regular expression operations](http://docs.python.org/2/library/re.html#module-re) says, "Regular expressions use the backslash character ('\') to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python’s usage of the same character for the same purpose in string literals..."

If you want a Python regular expression object which matches a newline character, then you need a 2-character string, consisting of the backslash character followed by the n character. The following lines of code all set prog to a regular expression object which recognises a newline character:

prog = re.compile(chr(92)+chr(110))

prog = re.compile("\\n")

prog = re.compile(r"\n")

So why is it that ["Usually patterns will be expressed in Python code using this raw string notation."](http://docs.python.org/2/library/re.html#module-re)? Because regular expressions are frequently static strings, which are conveniently represented as string literals. And from the different string literal notations available, raw strings are a convenient choice, when the regular expression includes a backslash character.