Splunk® Enterprise
Knowledge Manager Manual 9.0.4

About Splunk regular expressions

Generated: 2/14/2023 8:06 pm

About Splunk regular expressions

This primer helps you create valid regular expressions. For a discussion of regular expression syntax and usage, see an online resource such as www.regular-expressions.info or a manual on the subject.

Regular expressions match patterns of characters in text and are used for extracting default fields, recognizing binary file types, and automatic assignation of source types. You also use regular expressions when you define custom field extractions, filter events, route data, and correlate searches. Search commands that use regular expressions include `rex` and `regex` and evaluation functions such as `match` and `replace`.

Splunk regular expressions are PCRE (Perl Compatible Regular Expressions) and use the PCRE C library.

The Splunk platform includes the license for PCRE2, an improved version of PCRE. However, the Splunk platform does not currently allow access to functions specific to PCRE2, such as key substitution.

Regular expressions terminology and syntax

Term	Description
literal	The exact text of characters to match using a regular expression.
regular expression	The metacharacters that define the pattern that Splunk software uses to match against the literal.
groups	Regular expressions allow groupings indicated by the type of bracket used to enclose the regular expression characters. Groups can define character classes, repetition matches, named capture groups, modular regular expressions, and more. You can apply quantifiers to and use alternation within enclosed groups.
character class	Characters enclosed in square brackets. Used to match a string. To set up a character class, define a range with a hyphen, such as `[A-Z]`, to match any uppercase letter. Begin the character class with a caret (^) to define a negative match, such as `[^A-Z]` to match any lowercase letter.
character type	Similar to a wildcard, character types represent specific literal matches. For example, a period `.` matches any character, `\w` matches words or alphanumeric characters including an underscore, and so on.
anchor	Character types that match text formatting positions, such as return (`\r`) and newline (`\n`).
alternation	Refers to supplying alternate match patterns in the regular expression. Use a vertical bar or pipe character (|) to separate the alternate patterns, which can include full regular expressions. For example, `grey|gray` matches either `grey` or `gray`.
quantifiers, or repetitions	Use (`*, +, ?`) to define how to match the groups to the literal pattern. For example, `*` matches 0 or more, `+` matches 1 or more, and `?` matches 0 or 1.
back references	Literal groups that you can recall for later use. To indicate a back reference to the value, specify a dollar symbol (`$`) and a number (not zero).
lookarounds	A way to define a group to determine the position in a string. This definition matches the regular expression in the group but gives up the match to keep the result. For example, use a lookaround to match `x` that is followed by `y` without matching `y`.

Character types

Character types are short for literal matches.

Term	Description	Example	Explanation
`\w`		`\w\w\w`	Matches any three word characters.

Term	Description	Example	Explanation
	Match a word character (a letter, number, or underscore character).		
`\W`	Match a non-word character.	`\W\W\W`	Matches any three non-word characters.
`\d`	Match a digit character.	`\d\d\d-\d\d-\d\d\d\d`	Matches a Social Security number, or a similar 3-2-4 number string.
`\D`	Match a non-digit character.	\D\D\D	Matches any three non-digit characters.
`\s`	Match a whitespace character.	`\d\s\d`	Matches a sequence of a digit, a whitespace, and then another digit.
`\S`	Match a non-whitespace character.	`\d\S\d`	Matches a sequence of a digit, a non-whitespace character, and another digit.
`.`	Match any character. Use sparingly.	`\d\d.\d\d.\d\d`	Matches a date string such as 12/31/14 or 01.01.15, but can also match 99A99B99.

Groups, quantifiers, and alternation

Regular expressions allow groupings indicated by the type of bracket used to enclose the regular expression characters. You can apply quantifiers (`*, +, ?`) to the enclosed group and use alternation within the group.

Term	Description	Example	Explanation
`*`	Match zero or more times.	`\w*`	Matches zero or more word characters.
`+`	Match one or more times.	`\d+`	Match at least one digit.
`?`	Match zero or one time.	`\d\d\d-?\d\d-?\d\d\d\d`	Matches a Social Security Number with or without dashes.
`( )`	Parentheses define match or capture groups, atomic groups, and lookarounds.	`(H..).(o..)`	When given the string `Hello World`, this matches `Hel` and `o W`.
`[ ]`	Square brackets define character classes.	`[a-z0-9#]`	Matches any character that is `a` through `z`, `0` through `9`, or `#`.
`{ }`	Curly brackets define repetitions.	`\d{3,5}`	Matches a string of 3 to 5 digits in length.
`< >`	Angle brackets define named capture groups. Use the syntax `(?P<var> ...)` to set up a named field extraction.	`(?P<ssn>\d\d\d-\d\d-\d\d\d\d)`	Pulls out a Social Security Number and assigns it to the `ssn` field.
`[[ ]]`	Double brackets define Splunk-specific modular regular expressions.	`[[octet]]`	A validated 0-255 range integer.

A simple example of groups, quantifiers, and alternation

This example shows two ways to match either `to` or `too`.

The first regular expression uses the `?` quantifier to match up to one more "o" after the first.

The second regular expression uses alternation to specify the pattern.

```
to(o)?
```

`(to|too)`

Capture groups in regular expressions

A named capture group is a regular expression grouping that extracts a field value when regular expression matches an event. Capture groups include the name of the field. They are notated with angle brackets as follows:

```
matching text (?<field_name>capture pattern) more matching text.
```

For example, you have this event text:

```
131.253.24.135 fail admin_user
```

Here are two regular expressions that use different syntax in their capturing groups to pull the same set of fields from that event.

- Expression A: `(?<ip>\d+\.\d+\.\d+\.\d+) (?<result>\w+) (?<user>.*)`
- Expression B: `(?<ip>\S+) (?<result>\S+) (?<user>\S+)`

In Expression A, the pattern-matching characters used for the first capture group (`ip`) are specific. `\d` means "digit" and `+` means "one or more." So `\d+` means "one or more digits." `\.` refers to a period.

The capture group for `ip` wants to match one or more digits, followed by a period, followed by one or more digits, followed by a period, followed by one or more digits, followed by a period, followed by one or more digits. This describes the syntax for an ip address.

The second capture group in Expression A for the `result` field has the pattern `\w+`, which means "one or more alphanumeric characters." The third capture group in Expression A for the `user` field has the pattern `.*`, which means "match everything that's left."

Expression B uses a common technique called negative matching. With negative matching, the regular expression does not try to define which text to match. Instead it defines what the text is not. In this Expression B, the values that should be extracted from the sample event are "not space" characters (`\S`). It uses the `+` to specify "one or more" of the "not space" characters.

So Expression B says:

1. Pull out the first string of not-space characters for the `ip` field value.
2. Ignore the following space.
3. Then pull out the second string of not-space characters for the `result` field value.
4. Ignore the second space.
5. Pull out the third string of not-space characters for the `user` field value."

Non-capturing group matching

Use the syntax `(?: ... )` to create groups that are matched but which are not captured. Note that here you do not need to include a field name in angle brackets. The colon character after the `?` character is what identifies it as a non-capturing group.

For example, `(?:Foo|Bar)` matches either `Foo` or `Bar`, but neither string is captured.

Modular regular expressions

Modular regular expressions refer to small chunks of regular expressions that are defined to be used in longer regular expression definitions. Modular regular expressions are defined in transforms.conf.

For example, you can define an integer and then use that regular expression definition to define a float.

```
[int]
# matches an integer or a hex number
REGEX = 0x[a-fA-F0-9]+|\d+

[float]
# matches a float (or an int)
REGEX = \d*\.\d+|[[int]]
```

In the regular expression for `[float]`, the modular regular expression for an integer or hex number match is invoked with double square brackets, `[[int]]`.

You can also use the modular regular expression in field extractions.

```
[octet]
# this would match only numbers from 0-255 (one octet in an ip)
REGEX = (?:2(?:5[0-5]|[0-4][0-9])|[0-1][0-9][0-9]|[0-9][0-9]?)

[ipv4]
# matches a valid IPv4 optionally followed by :port_num the
# octets in the ip would also be validated 0-255 range
# Extracts: ip, port
REGEX = (?<ip>[[octet]](?:\.[[octet]]){3})(?::[[int:port]])?
```

The `[octet]` regular expression uses two nested non-capturing groups to do its work. See the subsection in this topic on non-capturing group matching.