## Things To Remember

* Different languages (C++, C#, JavaScript, Python, etc.) process Regex in **slightly different ways**. So a correct regex in C# doesn't mean that it will be also correct in JavaScript.
* Best online **tools to check** regex: [www.regex101.com](http://www.regex101.com) (for the most accurate result for a specific language, use another online tool for this language).
* **Using Regex helps reduce execution time (in most cases), lines of code and bugs** compared to processing strings manually. In case of having bugs, Regex helps fix them easily because in most cases all we have to do is modifying the patterns; without Regex, we have to fix our code's logic.
* Regex patterns are processed from left side to right side. So always **think and type Regex pattern from left to right**.
* For characters which Regex defines as special ones (e.g. ^.[$()\*+?{\), you must push an **escape character** "\" before them so that these characters can be treated like usual ones.
* In some languages, the **compiler gives warnings or even errors for using special characters** in string. In these cases, we need to use raw string so that these special characters can be treated like normal ones. For example, in C++ a raw string literal starts with R"(and ends in)" – e.g.: R"(^\w+)".
* When should not use regex? [Here](https://softwareengineering.stackexchange.com/questions/113237/when-you-should-not-use-regular-expressions).
* How to improve performance for regex? [Here](https://www.loggly.com/blog/regexes-the-bad-better-best/).
* Tips to get the most of regex: [here](https://pybit.es/mastering-regex.html) and [here](https://code.tutsplus.com/tutorials/advanced-regular-expression-tips-and-techniques--net-11011).

## Flags

A regex usually comes within this form /abc/, where the search pattern is delimited by two slash characters /. At the end, we can specify a flag with these following values (we can also combine them flexibly):

* **g** (global) does not return after the first match, restarting the subsequent searches from the end of the previous match.
* **i** (insensitive) makes the whole expression case-insensitive. E.g.: '/aBc/i' would match 'AbC'.
* **m** (multi-line) when enabled ^ and $ will match the start and end of a line, instead of the whole string
* **x** (ignore whitespace) eliminates unescaped white space from the pattern and enables comments marked with #. E.g.: '/12 3/x' would match '123'. More [here](http://xregexp.com/flags/).
* **…**

## Grouping

In regex, we use a pair of '()' bracket for grouping.

When using with most programming languages, any multiple occurrences captured by several groups will be exposed in the form of a classical **array**: we will access their values using an index on the result of the match.

Example:

|  |  |  |
| --- | --- | --- |
| **Description** | **Syntax** | **Example** |
| Match any 'abc' or 'bc'   * Full match: 'abc' * Group 1: 'bc' | a(bc) | ab abc abccccc babcbc |
|  |  |  |

## Table of Syntax

|  |  |  |
| --- | --- | --- |
| **Description** | **Syntax** | **Example** |
| **Anchors: ^ and $** | | |
| Match any 'house' | house | 1house2house. |
| Match any 'My' at the **beginning of string** | ^My | My\_house is beautiful. So is your house. |
| Match any 'house' at the **end of string** | house$ | My house is beautiful. So is your\_house |
| Match any whole phrase 'My house is', in which 'My' is at the beginning and 'is' is at the end | ^My house is$ | My house is  My house is…  My house is |
| /^My house is$/gm | My house is  My house is…  My house is |
| Match any thing to end of line  Note: Each line is a groupd   * Group 1: My house is * Group 2: null * Group 3: Your house is * Group 4: null | .\* | My house is  Your house is |
| **Character classes: \d \w \s and .** | | |
| Match a single character that is a **digit** | \d | abc a2b a42c |
| Match a single character that is a **digit**, **letter**, or **underscore** (NOT space, tab and symbol) | \w | ab 2 \_ -+=\/{}!@#$%^&\*(),./<>?;':"[] |
| Match a single character that is a **space** or **tab** or **line feed** (\n) or **carriage return** (\r) | \s | ab2 \_ - + |
| Match a single character that is a **tab** | \t | ab2 \_ - + |
| **Inverse** match of \d, \w and \s respectively | \D | ab2b 42c |
| \W | a 2 \_- |
| \S | a2 \_ - |
| Match '$' **followed by** a digit  Note: Use escape character '\' before '$'  List of characters that need '\' before are:  ^.[$()\*+?{\ | \$\d | It were sold for $1 |
| **OR operator: |** | | |
| Match any 'ab' or 'ac' | a(b|c) | abc ac acb aob |
| **Quantifiers: + ? \* and {}**  Also called greedy operators which expand the match as far as they can through the provided text. | | |
| Match any 'abc' **followed by** one or more 'c' | abc+ | ab abc abccccc babcd |
| Same as '\w' but for multiple characters (not just single character) | \w+ | a\_123] bc |
| Same as '\d' but for multiple digits (not just single digit) | \d+ | a\_123] bc |
| Match any 'abc' or 'ab' (meaning 'c' can be missed) | abc? | ab a abc abccccc |
| Match any 'abc' **followed by** 1 'c' | abc{2} | ab abc abccccc babcc |
| Match any 'abc' **followed by** 1 to 3 'c' | abc{2,4} | ab abc abccccc babcc |
| Match any 'abc' **followed by** 1 'c' or more | abc{2,} | ab abc abccccc babcc |
| Note: Check [grouping](#_30j0zll) first.  Match any <full match> which stands alone or **followed by** zero, one or more 'bc'   * Full match: 'a', 'abc', 'abc', 'abcbc' respectively * Group 1: 'bc' | a(bc)\* | ab abc abccccc babcbc |
| Like above, but now <group 1> is disabled; only <full match> is kept | a(?:bc)\* | ab abc abccccc babcbc |
| Match any 'a' or 'abc' **followed by** 1 to 4 'bc' | a(bc){2,5} | ab abc abccccc babcbc |
| **Greedy and Lazy match** | | |
| Match any character (except for line feed) | . | a\_1 ~^ |
| Match any character **inside a pair** of '<>' bracket, except \n and \r.  Note: It matches until the last '>' | <.+> | This is a <div> simple div</div> test |
| Match any character **inside a pair** of '<>' bracket  Note: It matches until the nearest '>' | <.+?> | This is a <div> simple div</div> test |
| Match any character **inside a pair** of '<>' bracket, but don't include the pair.  Note: It matches until the nearest '>' | (?<=<).+?(?=>) | This is a <div> simple div</div> test |
| **Look-ahead and Look-behind: (?=) and (?<=)** | | |
| Match any 'a' followed by 'bc' but do NOT include 'bc' in the match | a(?=bc) | ab abc abccccc babcbc |
| Match any 'bc' proceded by 'a' but do NOT include 'a' in the match  **Note**: In Python, you might get this error "*look-behind requires fixed-width pattern*". In this case, lookbehinds need to be alternated in a group (e.g. (?<=a|bc) should be rewritten as (?:(?<=a)|(?<=bc))). Details: [here](https://stackoverflow.com/a/40617321/14835442). | (?<=a)bc | ab abc abccccc babcbc |
| Combine the look-ahead and look-behind | (?<=a)bc(?=bc) | ab abc abccccc babcbc |
| **Negative Look-ahead and Look-behind** | | |
|  |  |  |
| **Square bracket:  []** | | |
| Inside bracket expressions, all special characters (including the backslash \) lose their special powers.  So we **don't apply the “escape rule”**. | | |
| Match any 'a' or 'b' or 'c' | [abc]  is same as  [a-c]  is same as  a|b|c | abc |
| Match any digit from '0' to '9' **followed by** '%' | [0-9]% | The 4nd month's percentage is 7%. |
| Match any <\w> or <\s>  Note: To only get space (' '), NOT tab or line feed or carriage return, use '**[\w ]+**' | [\w\s]+  WRONG: \w|\s | a\_123] bc d  de |
| Full match: Perotto, Pier Giorgio Batta  Group 1: Perotto  Group 2: Pier Giorgio | (\w+), ([\w]+[\s]+[\w]+) | Perotto, Pier Giorgio Batta |
| **Negation Operators: ^ and !** | | |
| Match any character, **except lower letters** ('a' to 'z').  Note: For excepting upper letters ('A' to 'Z') as well, use [^a-zA-Z] | [^a-z] | It were sold for $1 |
|  |  |  |
| **Boundaries: \b and \B** | | |
| Match any **whole word** 'abc' | \babc\b | ab abc abcc babcd |
| Match only if 'abc' is **fully surrounded** (both ends) by word characters | \Babc\B | ab abc abcc babcd |
| **Grouping: ()** | | |
| Match the **second occurrence** of the pattern | foo.\*?(foo) | foo, bar, foo |
| **Back-references — \1** | | |
|  |  |  |
| **Case insensity: (?i)** | | |
| Match any **whole word** 'abc' or 'ABC' or 'Abc' or … | \b(?i)abc\b | ab abc ABC babcd |
| **Which character to match?** | | |
| Match any character that is **letter** (case insensity), **digit**, **underscore**, **space**  **Note**:   * To allow more type of character, add it to []   E.g.: [**a-zA-Z0-9\_ !@#$%^&\*()]+** will match ab123 XYZ \_!@#$%^&\*()   * The + at the end means that there can be more than one characters, or a combination of characters.   To match only 0 to 2 characters, or a combination of characters, use **[a-zA-Z0-9\_ !@#$%^&\*()]{0,2}** | [a-zA-Z0-9\_ ]+ | ab123 XYZ \_!@#$%^&\*() |

## Regex Cookbook (Most Commonly Used)

<https://medium.com/factory-mind/regex-cookbook-most-wanted-regex-aa721558c3c1>

* Trim spaces
* HTML Tag
* Hexadecimal value
* Valid email (RFC5322)
* Username (simple)
* Strong password
* URL (http, https or ftp)
* IPv4 address
* “Defanged” URL or IPv4 address
* SSN — Social Security Number (simple)
* Alpha-numeric, literals, digits, lowercase, uppercase chars only