Regular Expressions

This cheat sheet provides a quick reference for essential regular expression (RegEx) constructs, helping you perform text pattern matching and manipulation with ease. It covers foundational syntax, such as character classes, anchors, and quantifiers, alongside advanced features like groups, lookaheads, and inline flags. Whether you're cleaning data, validating input, or performing complex text searches, this cheat sheet ensures you can find the right tools for the task.

Each entry includes practical examples that demonstrate regex's flexibility, from identifying patterns to modifying text with functions like re.sub(). The examples are paired with concise explanations to simplify learning and application. To test and refine your own patterns interactively, visit regex101.com, an indispensable tool for exploring how regex behaves.

Designed to make regex approachable and useful, this handy resource is perfect for tackling challenges in text processing, data cleaning, and parsing. Keep it close-by to be ready to streamline workflows and work effectively with text-based data.

Table of Contents

Special Characters

Syntax	Matches	Explained
`^r`	regular expressions	The `^` **anchor** matches the character or group to its right `r` only at the start of a string. It does not match if `r` appears elsewhere.
`s$`	she sells seashells	The `$` **anchor** matches the character or group to its left `s` only at the end of a string. It does not match if `s` appears elsewhere.
`.`	r e g u l a r e x p r e s s i o n s	The `.` **wildcard** matches any single character *(including spaces)* but not newline characters `\n`. It does not match multiple characters unless combined with a quantifier like `*` or `+`.
`\.`	www.example.com	The `\` character is used to escape special characters (e.g., `\.` for a literal dot) or to denote character classes (e.g., `\d` for digits). See the **Character Classes** section for more details.
`A|B`	Action Button	The `|` (OR) **operator** matches either the expression to its left `A` or its right `B`, finding all possible matches across the string.

Syntax	Matches	Explained
`b+`	a b c b b b d b	The `+` **quantifier** greedily matches the preceding expression `b` **one or more times**. It captures the longest possible sequences of `b` in the string.
`b*`	a b c b b b d b	The `*` **quantifier** greedily matches **zero or more** occurrences of the preceding expression `b`, including empty matches at positions where no `b` exists.
`colou?r`	color colour	The `?` **quantifier** matches the preceding character or group zero or one times, making it optional.
`u{3}`	uuu uuuu uu u	The `{m}` **quantifier** matches the preceding character `u` exactly `m` times.
`u{2,3}`	uuu uuuu uu u	The `{m,n}` **quantifier** matches the preceding character at least `m` times but not more than `n` times.
`u{2,3}?`	uuu uuuu uu u	The `{m,n}?` **quantifier** matches the preceding character at least `m` times but not more than `n` times, in a **non-greedy** (lazy) manner.

Sets

Syntax	Matches	Explained
`[rEsz]`	Regular Expression	Square brackets `[]` define a **set**, where each character is matched independently. A match occurs if any character from the set appears in the text.
`[a-z]`	1 Fig, 2 NewTons	The `-` in `[m-n]` is a **range operator**, matching any character from `m` to `n` .
`[a\-z]`	a to z is not = A-Z	The `\` escapes the `-` , treating it as a literal character instead of a range operator. This set matches `a` , `z` , and `-` only.
`[a-]`	regular-expression	Matches `a` and the literal `-` because `-` is treated as a character when placed at the start or end of a set.
`[-a]`	regular-expression	As above, matches `a` or `-` .
`[a-z0-9]`	396 ExpressionS	Matches characters from `a` to `z` and also from `0` to `9` .
`[(+*)]`	(valid) *expressions+words	Special characters become literal inside a set, so this matches `(` , `+`, `*`, and `)`.
`[^ers]`	regular expression	The `^` negates the set, matching any character not in the set. Here, it matches characters that are not `e` , `r` , or `s` .

Character Classes

Syntax	Matches	Explained
`\w`	Ch4racter_Class3s	Matches all alphanumeric characters (`a-z` , `A-Z` , and `0-9`). It also matches the underscore `_` .
`\W`	!@# $%^&*()	Matches any non-word character, which includes symbols, punctuation, and spaces. Non-word characters are anything not in the set `[a-zA-Z0-9_]`.
`\d`	1a2b3c	Matches all digits `0-9` .
`\D`	1a2b3c	Matches any non-digits.
`\s`	character classes	Matches whitespace characters including the `\t` , `\n` , `\r` , and space characters.
`\S`	character classes	Matches non-whitespace characters.
`\b`	character classes	Matches a word boundary, the position between a `\w` character (letter, digit, or underscore) and a `\W` character (non-word character). It doesn't match actual characters but positions like the start or end of words.
`\B`	character classes	Matches where `\b` does not, that is, the boundary of `\w` characters.
`\Ac`	color colour	Matches the start of the string. The backslash `\` escapes the normal meaning of `A`, turning it into a special positional anchor. Unlike `^`, which matches the start of each line in multi-line mode, `\A` always matches the very beginning of the entire string.
`r\Z`	color colour	Matches the end of the string. The backslash `\` escapes the normal meaning of `Z`, turning it into a special positional anchor. Unlike `$`, which matches the end of each line in multi-line mode, `\Z` always matches the very end of the entire string, excluding any trailing newline.

Popular Python `re` Module Functions

Syntax	Explained
`re.findall(A, B)`	Finds all non-overlapping matches of the pattern `A` in string `B` and returns them as a list. If no matches are found, it returns an empty list.
`re.search(A, B)`	Searches string `B` for the first occurrence of the pattern `A` and returns a match object. If no match is found, it returns `None`
`re.split(A, B)`	Splits string `B` into a list at each occurrence of the pattern `A` If no match is found, it returns the original string as a single-element list.
`re.sub(A, B, C)`	Replaces all occurrences of the pattern `A` in string `C` with the string `B` and returns the modified string. The original string `C` remains unchanged.
`re.match(A, B)`	Attempts to match the pattern `A` starting strictly at position `0` in string `B`. If the pattern doesn't match at the start, it returns `None`. Unlike `re.search()`, it does not evaluate the rest of the string.

Note: The `re` module is a part of Python's standard library so it does not need to be installed.
Run `import re` to access these functions.

Groups

Syntax	Matches	Explained
`(ro)`	groups	Captures the substring `ro` as a group. Groups are denoted by parentheses `()` and can be accessed later for further processing.
`(?:cat)`	cat fish dog	A **non-capturing group** groups patterns without creating a capturing group. Use non-capturing groups when grouping is needed for logic but you don't need to extract the group.
`cat(?=fish)`	catfish catdog	A **positive lookahead** asserts that the pattern `fish` must follow `cat` for a match. It checks the context after the current match without consuming it.
`(?<=cat)fish`	catfish dogfish	A **positive lookbehind** asserts that the pattern `cat` must precede `fish` for a match. It checks the context before the current match without consuming it.
`cat(?!fish)`	catfish catdog	A **negative lookahead** asserts that the pattern `fish` must not follow `cat` for a match. It checks the context after the current match without consuming it.
`(?<!cat)fish`	catfish dogfish	A **negative lookbehind** asserts that the pattern `cat` must not precede `fish` for a match. It checks the context before the current match without consuming it.
`(cat)\1`	catcat dogcat	The **backreference construct** `\1` refers to the first captured group in the pattern. Subsequent groups can be referenced with `\2`, `\3`, and so on.

Groups

Syntax	Matches	Explained
`(?P<pet>cat)`	dog cat fish	The **named group construct** `(?P<name>...)` assigns a `name` to the captured group for easy reference later in the regex. The `P` in `?P` stands for **Python**.
`(?P=pet)`	dog cat fish	The **named group backreference** `(?P=name)` matches the content previously captured by the named group `name`. In this example, it matches the word `cat`.
`(?#...)`		The **comment construct** allows you to include comments in your regex. These comments are ignored by the regex engine and do not affect the match result.

Inline Flags

Syntax	Matches	Explained
`(?aiLmsux)`		The **inline flag setting construct** `(?flags)` applies one or more flags to modify the behavior of the regex pattern that follows it. Use `(?flags:regex)` for group matching with flags.
`(?a)\w+`	cat 123_ CAT	The **ASCII-only flag** `a` restricts shorthand character classes like `\w`, `\W`, `\b`, and `\B` to match only ASCII characters, excluding Unicode.
`(?i)cat`	cat Cat CAT CaT	The **ignore case flag** `i` makes the pattern case-insensitive, allowing matches regardless of capitalization.
`(?L)\w+`	straße cafe Éclair	The **locale-dependent flag** `L` makes shorthand character classes like `\w` locale-sensitive, allowing matches based on cultural or regional rules.
`(?m)^cat`	catdog catfish	The **multi-line flag** `m` makes `^` and `$` match the start and end of each line, rather than the start and end of the entire string.
`(?s)cat.dog`	cat\ndog	The **dot matches all flag** `s` allows the `.` character to match newline characters in addition to all other characters.
`(?u)\w+`	naïve cat café	The **Unicode flag** `u` makes shorthand classes like `\w`, `\W`, `\b`, and `\B` match Unicode characters. Unlike the `L` flag, which applies locale-specific rules, the `u` flag uses Unicode rules to ensure consistent matching across languages.
`(?x)c a t`	cat	The **verbose flag** `x` enables e**x**tended formatting by allowing spaces and comments in the pattern for improved readability. Spaces are ignored unless escaped with a backslash.