Em Rainer Regex Docs (Python)

“Word” characters are: uppercase letters, lowercase letters, digits, and \_

Whitespace characters are: spaces, \n, return, and \t (tab)

Punctuation characters are kind of neither.

CASE MATTERS IN REGEX

You must use **“import re”** to use regex in python

Regex code in python generally returns match objects. These cannot be printed normally.

Single Character Match Patterns:

| Does: | Syntax: |
| --- | --- |
| Matches any character other than \n. Overrides the matching of periods, so periods need to be escaped to be specifically matched. | . |
| Matches one word character | \w |
| Matches non-word characters | \W |
| “Boundary between word and non-word.” Matches the empty string (zero width word boundary) at the beginning or end of a word. **Essentially matches the locations where word and whitespace characters meet.** | \b |
| Seems to match the boundary between anything that isn’t the beginning/end of a word. Ignores the intersections between word characters and whitespace characters. | \B |
| Matches one whitespace character | \s |
| Matches any character that isn’t whitespace | \S |
| Tab in regex | \t |
| New line (not rocket science, but ok) (\n  Essentially hits enter and starts another line) | \n |
| Return in regex (MAY BE INCORRECT: a \r returns of starting point of same line and deletes previously printed portion) | \r |
| A control character (the key, not the game) | \c |
| A lowercase character (changes the next letter to lowercase) | \l |
| Makes all characters to the end of the quote lowercase | \L |
| An uppercase character (changes next character to uppercase) | \u |
| Makes all characters up until the end of the quote uppercase | \U |
| A vertical tab. Essentially skip to the next line but keep previous line’s position. Pretty cool, but normally used for printers. Also used for soft wrapping in microsoft word (shift enter). Also can help scroll console screen quickly if supported (glaring at you, pycharm & replit.) | \v |
| A form feed character (essentially the opposite of a vertical tab. Goes up while keeping line position) | \f |
| A backspace character (why?) | [\b] |
| Escapes an octal character | \octalNumber |
| Escapes a hexadecimal character | \x\_\_\_\_\_\_\_\_ |
| Matches a decimal (numerical) digit | \d |
| Matches all non decimal digits | \D |
| Matches start of string | ^ |
| Matches end of string. Can be used to dictate a hard end to a pattern. | $ |
| Removes a character’s special functions (such as ., ^, etc.) If you want to look for characters such as \, this is REQUIRED. | \ |
| Creates a list of options for a specifically placed character to be. Put a ^ before the letters to insead make them exclusions. Put a dash between letters to make them a range. | [letters in here] or [^letters in here]  [a-z] (letters from a to z) |
| Put curly braces after anything (character, bracket, special character) to dictate num of times that bracket’s character can appear (sequentially, top of range is inclusive.) | \s{#} (specific num), \s{#, #} (specific range), or \s{#,} (specific num or more) |
| Means that whatever’s before it can repeat anywhere from 0 to infinity times sequentially. If after a grouping, applies to the entire grouped pattern. | .\* |
| Means that whatever’s before it can repeat anywhere from 1 to infinity times sequentially. Essentially means it has to be included. If after a grouping, applies to the entire grouped pattern. | .+ |
| Makes the character before optional. Must be escaped if you want to match it in the text. This also denotes greed, but that’ll be in a separate table. | .? |
| Put parenthesis around a specific part of regex to only return the match of the information inside the parenthesis. Can be nested, and if nested, matches information in exterior parenthesis order. | (info) or (in(fo)) |
| Or pipe can be put in regex to denote either\or | | |
| (May not work in all regex systems) allows you to reference captured match groups after capture. 0 is everything matched, from there group 1 and higher can be referenced. | \# (# = number of group in capture order) |

Nongreedy expressions:

Nongreedy quantifiers match as little as possible compared to normal quantifiers. Regex is greedy by default. For instance, if you had the base string <text>moretext> and asked regex for the quotes and everything in between (perhaps with something like **<.\*>**), regex would return <text>moretext>. This is because it always attempts to return the longest match possible for your query. However, if you told it to be nongreedy, aka placed a question mark behind the operator (**<.\*?>**), regex would instead return <text>. Nongreedy operators denote that regex must use a different ruleset for matching text for this instance of the quantifier. There’s a good explanation for it on this site: <https://blog.finxter.com/python-regex-greedy-vs-non-greedy-quantifiers/>

It really doesn’t matter too much, but non-greedy can be faster than greedy matches if you’re doing a single match search.

| Does:  (shoddy understanding, may be wrong) | Syntax: |
| --- | --- |
| Normal ? in regex matches the character(s in the case of a [] or group), but will match an instance with the character whenever possible. ?? instead tries to match the version without the character unless it’s “forced” to match the version with it. | .?? |
| Normal \* finds a match of a pattern that can repeat anywhere from 0 to infinity times, but matches the pattern in its entirety. In comparison, \*? prefers to return the zero match, but won’t if forced. | \*? |
| Pretty much the same as above. Will return the match with the least number of characters, if possible. | +? |
| Do more on this later | {#}? or {#,}? or {#,#}? |

Compilation Flags (Dictate rules for the compiler to follow when parsing request):

| Does: | Syntax: |
| --- | --- |
| (Expain better) Makes escapes such as \w, \b, \s, \d, etc. only work with ASCII characters with the property | ASCII or A (alias re.A) (inline ?a) |
| Allows . to also match a newline character | DOTALL or S (alias re.S) (inline ?s) |
| Makes matches not care about case | IGNORECASE or I (alias re.I) (inline ?i) |
| (Explain better) Forces a locale-aware match, aka treats all characters that are considered letters in your locale settings as /w characters. | LOCALE or L (alias re.L) (inline ?L) |
| Enables multi-line matching for ^ and $, aka makes them match the end/beginning of all lines, not just the start/end of the string. | MULTILINE or M (alias re.M) (inline ?m) |
| Enables verbose regular expressions, which are more understandable. Essentially ignores all whitespace characters that aren’t inside a set or escaped with a backslash. Also treats unescaped # as a comment marker. | VERBOSE or X (alias re.X) (inline ?x) |
| Interprets letters according to the unicode character set, aka treats all letters from all scripts as word (\w) characters. Affects the behavior of \w, \W, \b, \B. | UNICODE or U (alias re.U) (no inline) |
| Displays debug information about compiled expression | DEBUG (no alias) (no inline) |

Lookaround Assertions:

0 length assertions that are either placed at the beginning or end of a request or group. They match characters, but then drop the match and return a boolean result that the rest of the regex can utilize. They do not alter the characters in the given string or advance the regex’s position in the string. They’re “looking”, not “moving.” They’re meant to assert whether a match is possible.

Be careful when using lookaheads. Since they are 0 length assertions, regex may try to match whatever is placed directly after them to the same position. Try putting a . after them if you want to look for letters after a must or must not equal lookahead. Think about it like they’re assertions about the character that precedes them, not about the characters after it. Any valid regular expression can be included in a lookahead, but not in a lookbehind.

A lookbehind tells the regex engine to step backwards in a string to validate it. It can only do fixed length strings. This means you can do literal text, escaped characters, unicode escapes that aren’t \X, and {} in python. Java is a bit more flexible, as it allows for the use of ? and {}. In java 13, you can even use \* and +. However, these can be prone to error and incorrect matches.

Regex only works with the first occurrence of the lookaround. It ignores other permutations. This can make it hard to use capturing groups inside lookarounds, which you *can* do.

| Does: | Syntax: |
| --- | --- |
| Must be followed by (lookahead) | (?= thingThatMustBeMatched) |
| Must not be followed by (lookahead) | (?! thingThatMustNotBeMatched) |
| Dictates that the following group will not be captured (neutral) | (?: things) |
| Calls a compilation flag for the text after it (neutral) | (?inlineAlias) |
| Defines a comment in regex (learn) | (?# moreStuff) |
| Names a capture group, which allows them to be referenced later (neutral, learn more) | ?’groupName’  To call groupName: k{group} |
| Must be preceded by (lookbehind) | (?<= text) |
| Must not be preceded by (lookbehind) | (?<! text) |

Match Function:

**re.match(pattern, string, flags)**

Attempts the pattern at the start of the string only. Essentially the same as passing \A\_\_\_.

Returns a match object on success, None on failure

Ex: re.match(r’.’, "I do not have a cat but please, take my cat?", flags = 0) returns “I”

| Pattern: | String: | Flags: |
| --- | --- | --- |
| The regular expression to be matched | The string to be searched | The place to put compilation flags. Flags are optional in this command. Use flags = 0 to pass none. To use multiple flags, put a pipe (|) or an OR between them |

Full Match Function:

**re.fullmatch(pattern, string, flags)** (Same as above)

Essentially the same as passing /A\_\_\_\_/B

Only returns a match object if the regex matches the entire string, returns None if not there.

Ex: re.fullmatch(r'.', "I", flags = 0) returns “I” because it matches the full string

Search Function:

**re.search(pattern, string, flags)** (Same as above)

Searches entire string, not just the start.

Returns a match object if regex defined string is in given string somewhere, None on failure.

Ex: re.search(r'.', "A E I O U", flags = 0) returns “A”, because it’s the first location that a character exists.

Find All Function: (*Does not allow for flags to be called, must use aliases to put them in regex*)

**re.findall(pattern, string)** (Pretty much same as above)

Searches entire string for all instances that match pattern.

Returns one or more (in list) results, None if nothing found. Does not need group(s) to print.

Ex: re.findall(r'\w', "A E I O U", flags = 0) returns ['A', 'E', 'I', 'O', 'U'].

Find Iterator Function: (*Same as find all, different return type*)

**re.finditer(pattern, string)** (identical to above)

Searches entire string for all instances that match pattern.

Returns one or more (in iterator object) results in match object, None if nothing found. Needs for each loop and group(s).

Ex: re.finditer(r'\w', "A E I O U", flags = 0) returns an iterator with match objects that contain 'A', 'E', 'I', 'O', and 'U'.

Split Function:

**re.split(regex, subject)** (*Also identical to find all*)

Returns an array of strings containing the parts between regex matches in the subject. Empty strings are included in this, if possible to return them.

Ex: re.split(r'[Rr]ed', "Red is a red is a red is a red.") returns ['', ' is a ', ' is a ', ' is a ', '.']

Search and Replace Function:

**re.sub(pattern, replacement pattern, string, count (optional), flags)**

Does not modify input string. Finds all instances of a pattern and replaces said pattern with replacement pattern.

Returns a string with the applied changes.

Ex: re.sub(r'[Rr]ed', 'blue', "Red is a red is a red is a red.", count = 2, flags = 0) returns “blue is a blue is a red is a red.” because we asked the code to stop at 2 replacements.

| Pattern: | Replacement Pattern: | String: | Count: | Flags: |
| --- | --- | --- | --- | --- |
| The regex to search for. | Dictates what to replace all matches with. Can contain groups received from the first pattern if said groups are called by name or by match order number. | The base string. | The way to tell the program how many replacements to stop at. This is optional. | The place to put compilation flags. Flags are also optional in this command. Use flags = 0 to pass none. To use multiple flags, put a pipe (|) or an OR between them. |

To print a match object, use…

print(matchObject.group())

Returns entire group at once if empty or 0, returns specific groups by id (starts with 1) if filled.

OR

print(matchObject.groups())

Returns tuple of matched groups. Only really necessary if multiple groups were collected in the same regex call.

There are other ways to use a match object:

matchObject.start() returns the index of the location in the string of the start of the match.

matchObject.end() returns the index of the location of the character directly after the match.

matchObject.span() returns a 2-tuple of matchObject.start() and matchObject.end().

Start and end can be used for string slicing (baseString[m.start():m.end()])