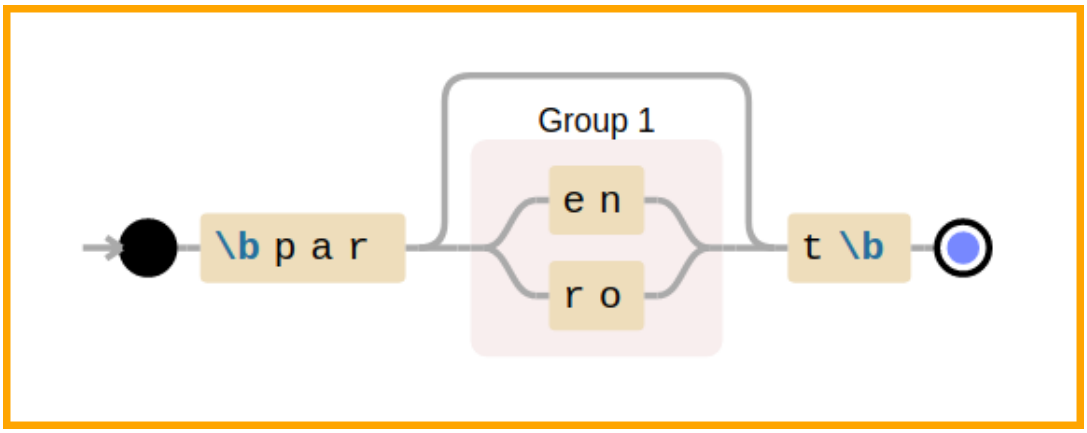
Python regular expression cheatsheet and examples

2020-07-03



Above visualization is a screenshot created using debuggex *for the pattern* `r'\bpar(en|ro)?t\b'`

From docs.python: re:

> 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

This blog post gives an overview and examples of regular expression syntax as implemented by the `re` built-in module (Python 3.8+). Assume ASCII character set unless otherwise specified. This post is an excerpt from my Python re(gex)? book.

Elements that define a regular expression🔗

Anchors	Description
`\A`	restricts the match to the start of string
`\Z`	restricts the match to the end of string
`^`	restricts the match to the start of line
`$`	restricts the match to the end of line

`\n`	newline character is used as line separator
`re.MULTILINE` or `re.M`	flag to treat input as multiline string
`\b`	restricts the match to the start/end of words
	word characters: alphabets, digits, underscore
`\B`	matches wherever `\b` doesn't match

`^`, `$` and `\` are metacharacters in the above table, as these characters have special meaning. Prefix a `\` character to remove the special meaning and match such characters literally. For example, `\^` will match a `^` character instead of acting as an anchor.

Feature	Description
`|`	multiple RE combined as conditional OR
	each alternative can have independent anchors
`(RE)`	group pattern(s), also a capturing group
	`a(b|c)d` is same as `abd|acd`
`(?:RE)`	non-capturing group
`(?P<name>pat)`	named capture group
`.`	Match any character except the newline character `\n`
`[]`	Character class, matches one character among many

Greedy Quantifiers	Description
`*`	Match zero or more times
`+`	Match one or more times
`?`	Match zero or one times
`{m,n}`	Match `m` to `n` times (inclusive)
`{m,}`	Match at least m times
`{,n}`	Match up to `n` times (including `0` times)
`{n}`	Match exactly n times
`pat1.*pat2`	any number of characters between `pat1` and `pat2`
`pat1.*pat2|pat2.*pat1`	match both `pat1` and `pat2` in any order

Greedy here means that the above quantifiers will match as much as possible that'll also honor the overall

RE. Appending a `?` to greedy quantifiers makes them **non-greedy**, i.e. match as *minimally* as possible. Quantifiers can be applied to literal characters, groups, backreferences and character classes.

Character class	Description
`[aeiou]`	Match any vowel
`[^aeiou]`	`^` inverts selection, so this matches any consonant
`[a-f]`	`-` defines a range, so this matches any of abcdef characters
`\d`	Match a digit, same as `[0-9]`
`\D`	Match non-digit, same as `[^0-9]` or `[^\d]`
`\w`	Match word character, same as `[a-zA-Z0-9_]`
`\W`	Match non-word character, same as `[^a-zA-Z0-9_]` or `[^\w]`
`\s`	Match whitespace character, same as `[\ \t\n\r\f\v]`
`\S`	Match non-whitespace character, same as `[^\ \t\n\r\f\v]` or `[^\s]`

Lookarounds	Description
lookarounds	custom assertions, zero-width like anchors
`(?!pat)`	negative lookahead assertion
`(?<!pat)`	negative lookbehind assertion
`(?=pat)`	positive lookahead assertion
`(?<=pat)`	positive lookbehind assertion
`(?!pat1)(?=pat2)`	multiple assertions can be specified in any order
	as they mark a matching location without consuming characters
`((?!pat).)*`	Negate a grouping, similar to negated character class

Flags	Description
`re.IGNORECASE` or `re.I`	flag to ignore case
`re.DOTALL` or `re.S`	allow `.` metacharacter to match newline character
`flags=re.S|re.I`	multiple flags can be combined using `|` operator
`re.MULTILINE` or `re.M`	allow `^` and `$` anchors to match line wise
`re.VERBOSE` or `re.X`	allows to use literal whitespaces for aligning purposes
	and to add comments after the `#` character

	escape spaces and `#` if needed as part of actual RE
`re.ASCII` or `re.A`	match only ASCII characters for `\b`, `\w`, `\d`, `\s`
	and their opposites, applicable only for Unicode patterns
`re.LOCALE` or `re.L`	use locale settings for byte patterns and 8-bit locales
`(?#comment)`	another way to add comments, not a flag
`(?flags:pat)`	inline flags only for this `pat`, overrides `flags` argument
	flags is `i` for `re.I`, `s` for `re.S`, etc, except `L` for `re.L`
`(?-flags:pat)`	negate flags only for this `pat`
`(?flags-flags:pat)`	apply and negate particular flags only for this `pat`
`(?flags)`	apply flags for whole RE, can be used only at start of RE
	anchors if any, should be specified after `(?flags)`

Matched portion	Description
`re.Match` object	details like matched portions, location, etc
`m[0]` or `m.group(0)`	entire matched portion of `re.Match` object `m`
`m[n]` or `m.group(n)`	matched portion of *n*th capture group
`m.groups()`	tuple of all the capture groups' matched portions
`m.span()`	start and end+1 index of entire matched portion
	pass a number to get span of that particular capture group
	can also use `m.start()` and `m.end()`
`\N`	backreference, gives matched portion of *N*th capture group
	applies to both search and replacement sections
	possible values: `\1`, `\2` up to `\99` provided no more digits
`\g<N>`	backreference, gives matched portion of Nth capture group
	possible values: `\g<0>`, `\g<1>`, etc (not limited to 99)
	`\g<0>` refers to entire matched portion
`(?P<name>pat)`	named capture group
	refer as `'name'` in `re.Match` object
	refer as `(?P=name)` in search section

	refer as `\g<name>` in replacement section
`groupdict`	method applied on a `re.Match` object
	gives named capture group portions as a `dict`

> ℹ️ `\0` and `\100` onwards are considered as octal values, hence cannot be used as backreferences.

re module functions🔗

Function	Description
`re.search`	Check if given pattern is present anywhere in input string
	Output is a `re.Match` object, usable in conditional expressions
	r-strings preferred to define RE
	Use byte pattern for byte input
	Python also maintains a small cache of recent RE
`re.fullmatch`	ensures pattern matches the entire input string
`re.compile`	Compile a pattern for reuse, outputs `re.Pattern` object
`re.sub`	search and replace
`re.sub(r'pat', f, s)`	function `f` with `re.Match` object as argument
`re.escape`	automatically escape all metacharacters
`re.split`	split a string based on RE
	text matched by the groups will be part of the output
	portion matched by pattern outside group won't be in output
`re.findall`	returns all the matches as a list
	if 1 capture group is used, only its matches are returned
	1+, each element will be tuple of capture groups
	portion matched by pattern outside group won't be in output
`re.finditer`	iterator with `re.Match` object for each match
`re.subn`	gives tuple of modified string and number of substitutions

The function definitions are given below:

```
re.search(pattern, string, flags=0)
re.fullmatch(pattern, string, flags=0)
re.compile(pattern, flags=0)
re.sub(pattern, repl, string, count=0, flags=0)
re.escape(pattern)
re.split(pattern, string, maxsplit=0, flags=0)
re.findall(pattern, string, flags=0)
re.finditer(pattern, string, flags=0)
re.subn(pattern, repl, string, count=0, flags=0)
```

Regular expression examples🔗

As a good practice, always use **raw strings** to construct RE, unless other formats are required. This will avoid clash of special meaning of backslash character between RE and normal quoted strings.

- examples for `re.search`

```
>>> sentence = 'This is a sample string'

# need to load the re module before use
>>> import re
# check if 'sentence' contains the pattern described by RE argument
>>> bool(re.search(r'is', sentence))
True
# ignore case while searching for a match
>>> bool(re.search(r'this', sentence, flags=re.I))
True
>>> bool(re.search(r'xyz', sentence))
False

# re.search output can be directly used in conditional expressions
>>> if re.search(r'ring', sentence):
...     print('mission success')
...
mission success

# use raw byte strings if input is of byte data type
>>> bool(re.search(rb'is', b'This is a sample string'))
True
```

- difference between string and line anchors

```
# string anchors
>>> bool(re.search(r'\Ahi', 'hi hello\ntop spot'))
True
```

```
>>> words = ['surrender', 'up', 'newer', 'do', 'ear', 'eel', 'pest']
>>> [w for w in words if re.search(r'er\Z', w)]
['surrender', 'newer']

# line anchors
>>> bool(re.search(r'^par$', 'spare\npar\ndare', flags=re.M))
True
```

- examples for `re.findall`

```
# whole word par with optional s at start and optional e at end
>>> re.findall(r'\bs?pare?\b', 'par spar apparent spare part pare')
['par', 'spar', 'spare', 'pare']

# numbers >= 100 with optional leading zeros
>>> re.findall(r'\b0*[1-9]\d{2,}\b', '0501 035 154 12 26 98234')
['0501', '154', '98234']

# if multiple capturing groups are used, each element of output
# will be a tuple of strings of all the capture groups
>>> re.findall(r'([^/]+)/([^/,]+),?', '2020/04,1986/Mar')
[('2020', '04'), ('1986', 'Mar')]

# normal capture group will hinder ability to get whole match
# non-capturing group to the rescue
>>> re.findall(r'\b\w*(?:st|in)\b', 'cost akin more east run')
['cost', 'akin', 'east']

# useful for debugging purposes as well
>>> re.findall(r't.*?a', 'that is quite a fabricated tale')
['tha', 't is quite a', 'ted ta']
```

- examples for `re.split`

```
# split based on one or more digit characters
>>> re.split(r'\d+', 'Sample123string42with777numbers')
['Sample', 'string', 'with', 'numbers']

# split based on digit or whitespace characters
>>> re.split(r'[\d\s]+', '**1\f2\n3star\t7 77\r**')
['**', 'star', '**']

# to include the matching delimiter strings as well in the output
>>> re.split(r'(\d+)', 'Sample123string42with777numbers')
['Sample', '123', 'string', '42', 'with', '777', 'numbers']

# use non-capturing group if capturing is not needed
>>> re.split(r'hand(?:y|ful)', '123handed42handy777handful500')
['123handed42', '777', '500']
```

- backreferencing within search pattern

```
# whole words that have at least one consecutive repeated character
>>> words = ['effort', 'flee', 'facade', 'oddball', 'rat', 'tool']

>>> [w for w in words if re.search(r'\b\w*(\w)\1\w*\b', w)]
['effort', 'flee', 'oddball', 'tool']
```

- working with matched portions

```
>>> re.search(r'b.*d', 'abc ac adc abbbc')
<re.Match object; span=(1, 9), match='bc ac ad'>
# retrieving entire matched portion, note the use of [0]
>>> re.search(r'b.*d', 'abc ac adc abbbc')[0]
'bc ac ad'

# capture group example
>>> m = re.search(r'a(.*)d(.*a)', 'abc ac adc abbbc')
# to get matched portion of second capture group
>>> m[2]
'c a'
# to get a tuple of all the capture groups
>>> m.groups()
('bc ac a', 'c a')
```

- examples for `re.finditer`

```
# numbers < 350
>>> m_iter = re.finditer(r'[0-9]+', '45 349 651 593 4 204')
>>> [m[0] for m in m_iter if int(m[0]) < 350]
['45', '349', '4', '204']

# start and end+1 index of each matching portion
>>> m_iter = re.finditer(r'ab+c', 'abc ac adc abbbc')
>>> for m in m_iter:
...     print(m.span())
...
(0, 3)
(11, 16)
```

- examples for `re.sub`

```
>>> ip_lines = "catapults\nconcatenate\ncat"
>>> print(re.sub(r'^', r'* ', ip_lines, flags=re.M))
* catapults
* concatenate
* cat

# replace 'par' only at start of word
```

```
>>> re.sub(r'\bpar', r'X', 'par spar apparent spare part')
'X spar apparent spare Xt'

# same as: r'part|parrot|parent'
>>> re.sub(r'par(en|ro)?t', r'X', 'par part parrot parent')
'par X X X'

# remove first two columns where : is delimiter
>>> re.sub(r'\A([^:]+:){2}', r'', 'foo:123:bar:baz', count=1)
'bar:baz'
```

- backreferencing in replacement section

```
# remove consecutive duplicate words separated by space
>>> re.sub(r'\b(\w+)( \1)+\b', r'\1', 'aa a a a 42 f_1 f_1 f_13.14')
'aa a 42 f_1 f_13.14'

# add something around the matched strings
>>> re.sub(r'\d+', r'(\g<0>0)', '52 apples and 31 mangoes')
'(520) apples and (310) mangoes'

# swap words that are separated by a comma
>>> re.sub(r'(\w+),(\w+)', r'\2,\1', 'good,bad 42,24')
'bad,good 24,42'
```

- using functions in replacement section of `re.sub`

```
>>> from math import factorial
>>> numbers = '1 2 3 4 5'
>>> def fact_num(n):
...     return str(factorial(int(n[0])))
...
>>> re.sub(r'\d+', fact_num, numbers)
'1 2 6 24 120'

# using lambda
>>> re.sub(r'\d+', lambda m: str(factorial(int(m[0]))), numbers)
'1 2 6 24 120'
```

- examples for lookarounds

```
# change 'foo' only if it is not followed by a digit character
# note that end of string satisfies the given assertion
# foofoo has 2 matches as the assertion doesn't consume characters
>>> re.sub(r'foo(?!\d)', r'baz', 'hey food! foo42 foot5 foofoo')
'hey bazd! foo42 bazt5 bazbaz'

# change whole word only if it is not preceded by : or -
>>> re.sub(r'(?<![:-])\b\w+\b', r'X', ':cart <apple -rest ;tea')
':cart <X -rest ;X'
```

```
# match digits only if it is preceded by - and followed by ; or :
>>> re.findall(r'(?<=-)\d+(?=[:;])', 'fo-5, ba3; x-83, y-20: f12')
['20']

# words containing 'b' and 'e' and 't' in any order
>>> words = ['sequoia', 'questionable', 'exhibit', 'equation']
>>> [w for w in words if re.search(r'(?=.*b)(?=.*e).*t', w)]
['questionable', 'exhibit']

# match if 'do' is not there between 'at' and 'par'
>>> bool(re.search(r'at((?!do).)*par', 'fox,cat,dog,parrot'))
False
# match if 'go' is not there between 'at' and 'par'
>>> bool(re.search(r'at((?!go).)*par', 'fox,cat,dog,parrot'))
True
```

- examples for `re.compile`

Regular expressions can be compiled using `re.compile` function, which gives back a `re.Pattern` object. The top level `re` module functions are all available as methods for this object. Compiling a regular expression helps if the RE has to be used in multiple places or called upon multiple times inside a loop (speed benefit). By default, Python maintains a small list of recently used RE, so the speed benefit doesn't apply for trivial use cases.

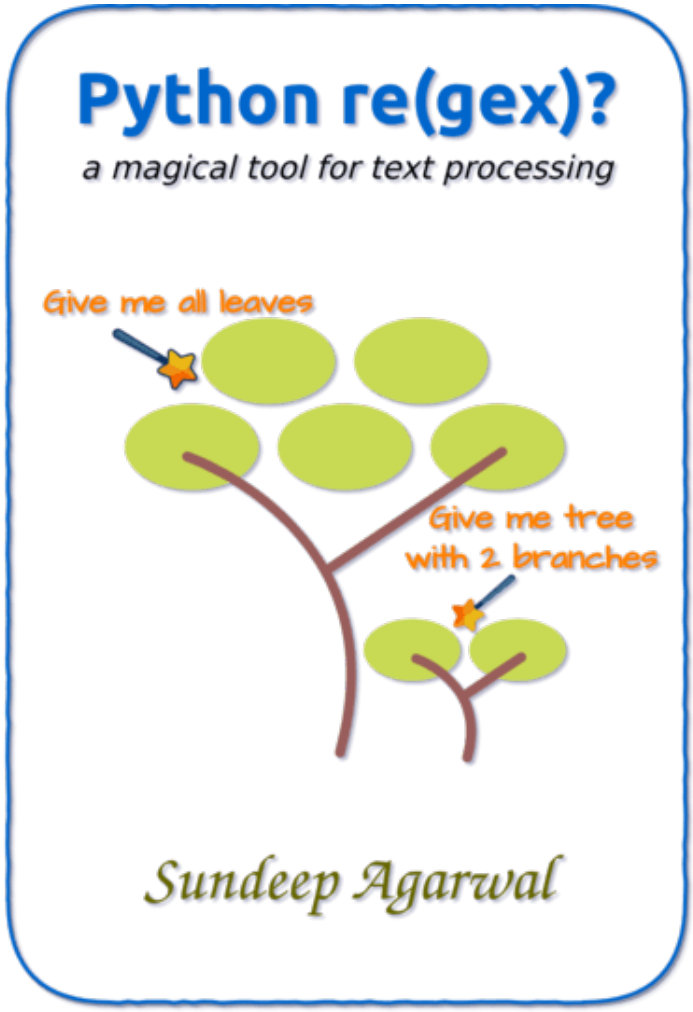
```
>>> pet = re.compile(r'dog')
>>> type(pet)
<class 're.Pattern'>
>>> bool(pet.search('They bought a dog'))
True
>>> bool(pet.search('A cat crossed their path'))
False

>>> pat = re.compile(r'\([^)]*\)')
>>> pat.sub('', 'a+b(addition) - foo() + c%d(#modulo)')
'a+b - foo + c%d'
>>> pat.sub('', 'Hi there(greeting). Nice day(a(b)')
'Hi there. Nice day'
```

Python re(gex)? book🔗

Visit my repo Python re(gex)? for details about the book I wrote on Python regular expressions. The ebook uses plenty of examples to explain the concepts from the very beginning and step by step introduces more advanced concepts. The book also covers the third party module regex. The cheatsheet and examples presented in this post are based on contents of this book.

Use this leanpub link for a discounted price.



#python #regular-expressions #cheatsheet #re-module #examples

‹ Example driven book on Python regular expressions

JavaScript regular expressions cheatsheet and examples ›