**Regular expressions- Used for Pattern matching:**

A regular expression is a sequence of letters and some special characters (also called

meta characters). These special characters have symbolic meaning.

The sequence formed by using meta characters and letters can be used to represent a group of patterns.

Usually Regular expression pattern is used by [string searching algorithms](https://en.wikipedia.org/wiki/String_searching_algorithm) for "find" or "find and replace" operations on [strings](https://en.wikipedia.org/wiki/String_(computer_science)).

Regular expressions can be used in python for matching a particular pattern by importing the re module.

The re module raises the exception re.error if an error occurs while compiling or using a regular expression.

# Raw Python strings

When writing regular expression in Python, it is recommended that you use [raw strings](https://docs.python.org/2/howto/regex.html#the-backslash-plague) instead of regular Python strings. Raw strings begin with a special prefix (r) and signal Python not to interpret backslashes and special metacharacters in the string, allowing you to pass them through directly to the regular expression engine.

This means that a pattern like "\n\w" will not be interpreted and can be written as r"\n\w" instead of "\\n\\w"

Raw String - **r'expression'-** may be written to avoid met characters.

It is important to note that most regular expression operations are available as module-level functions and methods on [compiled regular expressions](https://docs.python.org/3/library/re.html#re-objects).

**>>>** pattern = re.compile("d")

**>>>** pattern.search("dog") *# Match at index 0*

**For example**

str= “Ram$”

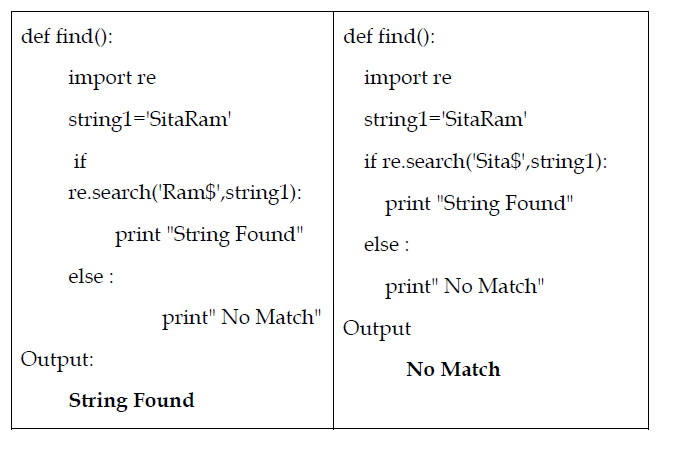
The pattern “Ram$” is known as a regular expression.

The expression has the metacharacter ‘$’. Meta character ‘$’ is used to match the given regular expression at the end of the string.

So the regular expression would match the string “SitaRam‟ or “HeyRam‟

but will not match the string “Raman‟.

More example:



**"or"**

A [**vertical bar**](https://en.wikipedia.org/wiki/Vertical_bar) separates alternatives. For example, gray|grey can match "gray" or "grey".

**Grouping**

[Parentheses](https://en.wikipedia.org/wiki/Bracket) are used to define the scope and precedence of the [operators](https://en.wikipedia.org/wiki/Operator_(programming)) (among other uses). For example, gray|grey and gr(a|e)y are equivalent patterns which both describe the set of "gray" or "grey".

**Quantification**

A [quantifier](https://en.wikipedia.org/wiki/Quantifier_(linguistics)) after a [token](https://en.wikipedia.org/wiki/Lexical_analysis#Token) (such as a character) or group specifies how often that preceding element is allowed to occur. The most common quantifiers are the [question mark](https://en.wikipedia.org/wiki/Question_mark) ?, the [asterisk](https://en.wikipedia.org/wiki/Asterisk) \* ,and the [plus sign](https://en.wikipedia.org/wiki/Plus_sign) + ([Kleene plus](https://en.wikipedia.org/wiki/Kleene_plus))

|  |  |
| --- | --- |
| **Quantifiers** | **Used for** |
| **?** | The question mark indicates *zero or one* occurrences of the preceding element. For example, colou?r matches both "color" and "colour". |
| **\*** | The asterisk indicates *zero or more* occurrences of the preceding element. For example, ab\*c matches "ac", "abc", "abbc", "abbbc", and so on. |
| **+** | The plus sign indicates *one or more* occurrences of the preceding element. For example, ab+c matches "abc", "abbc", "abbbc", and so on, but not "ac". |
| [{n}[19]](https://en.wikipedia.org/wiki/Regular_expression#cite_note-grep-19) | The preceding item is matched exactly *n* times. |
| [{min,}[19]](https://en.wikipedia.org/wiki/Regular_expression#cite_note-grep-19) | The preceding item is matched *min* or more times. |
| [{min,max}[19]](https://en.wikipedia.org/wiki/Regular_expression#cite_note-grep-19) | The preceding item is matched at least *min* times, but not more than *max* times. |

Above quantifiers can be combined to form arbitrarily complex expression.

**Meta Characters:**

|  |  |
| --- | --- |
| **Metachar** | **Description** |
| **.** | Matches any single character except a newline. If the DOTALL flag has been specified, this matches any character including a newline. For example, a.c matches "abc", etc., but [a.c] matches only "a", ".", or "c". |
| **[ ]** | A bracket expression. Matches a single character that is contained within the brackets. For example, [abc] matches "a", "b", or "c". [a-z] specifies a range which matches any lowercase letter from "a" to "z". These forms can be mixed: [abcx-z] matches "a", "b", "c", "x", "y", or "z", as does [a-cx-z]. |
| The - character is treated as a literal character if it is the last or the first (after the ^, if present) character within the brackets: [abc-], [-abc]. The ] character can be included in a bracket expression if it is the first (after the ^) character: []abc]. |
| **[^ ]** | Matches a single character that is not contained within the brackets. For example, [^abc] matches any character other than "a", "b", or "c". [^a-z] matches any single character that is not a lowercase letter from "a" to "z". Likewise, literal characters and ranges can be mixed. |
| **^** | Matches the starting position within the string. In line-based tools, it matches the starting position of any line. |
| **$** | Matches the ending position of the string or the position just before a string-ending newline. In line-based tools, it matches the ending position of any line. For E.g: Ram$ The regular expression would match Ram in SitaRam but will not match Ram in Raman |

**Escape Codes**

An even more compact representation uses escape codes for several pre-defined character sets. The escape codes recognized by [**re**](https://pymotw.com/2/re/#module-re) are:

| **Code** | **Meaning** |
| --- | --- |
| \d | a digit |
| \D | a non-digit |
| \s | whitespace (tab, space, newline, etc.) |
| \S | non-whitespace |
| \w | alphanumeric |
| \W | non-alphanumeric |

**Functions of ‘re’ module:**

* **re.search()**

re.**search**(*pattern*, *string*, *flags=0*)

Scan through *string* looking for the occurrence location where the regular expression *pattern* produces a match, and return a corresponding [match object](https://docs.python.org/3/library/re.html#match-objects). Return None if no position in the string matches the pattern;

The search function traverses through the string and determines the position where the

RE matches the string

**Example**

>>> m=re.search('hell\*o', 'favorite words hellooooo world')

>>> m.start()

15

>>> m.end()

20

>>> m.group()

'hello'

>>> m.span()

(15, 20)

* **re.match()**

The match function is used to determine if the regular expression (RE) matches at the

beginning of the string.

re.**match**(*pattern*, *string*, *flags=0*)

If zero or more characters at the beginning of *string* match the regular expression *pattern*, return a corresponding [match object](https://docs.python.org/3/library/re.html#match-objects). Return None if the string does not match the pattern;

search() vs. match()

Python offers two different primitive operations based on regular expressions: [re.match()](https://docs.python.org/3/library/re.html#re.match) checks for a match only at the beginning of the string, while[re.search()](https://docs.python.org/3/library/re.html#re.search) checks for a match anywhere in the string

For example:

>>>

**>>>** re.match("c", "abcdef") *# No match*

**>>>** re.search("c", "abcdef") *# Match*

Regular expressions beginning with '^' can be used with [search()](https://docs.python.org/3/library/re.html#re.search) to restrict the match at the beginning of the string:

>>>

**>>>** re.match("c", "abcdef") *# No match*

**>>>** re.search("^c", "abcdef") *# No match*

**>>>** re.search("^a", "abcdef") *# Match*

* **re.group()**

The group function is used to return the string matched the RE

**Example**

>>>P=re.compile(“hell\*o”)

>>>m=re.match(“hell\*o”, “hellooooo world”)

>>>m.group()

“hello”

**>>>** m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")

**>>>** m.group(0) *# The entire match*

'Isaac Newton'

**>>>** m.group(1) *# The first parenthesized subgroup.*

'Isaac'

**>>>** m.group(2) *# The second parenthesized subgroup.*

'Newton'

**>>>** m.group(1, 2) *# Multiple arguments give us a tuple.*

('Isaac', 'Newton')

* **re.start()**

returns the starting position of the match.

* **re.end()**

returns the end position of the match.

* **re.span()**

returns the tuple containing the (start, end) positions of the match.

**Example**

>>> import re

>>> P=re.compile('hell\*o')

>>> m=re.match('hell\*o', 'hellooooo world')

>>> m.start()

0

>>> m.end()

5

>>> m.span()

(0, 5)

* **Re.findall()**

[findall()](https://docs.python.org/3/library/re.html#re.findall) matches *all* occurrences of a pattern, not just the first one as [search()](https://docs.python.org/3/library/re.html#re.search) does. For example, if one was a writer and wanted to find all of the adverbs in some text, he or she might use [findall()](https://docs.python.org/3/library/re.html#re.findall) in the following manner:

>>>

**>>>** text = "He was carefully disguised but captured quickly by police."

**>>>** re.findall(r"\w+ly", text)

['carefully', 'quickly']

**Example**

>>> m=re.findall('hell\*o', 'hello my favorite words hellooooo world')

>>> m

['hello', 'hello']

* Finding all Adverbs and their Positions - **re.finditer()**

If one wants more information about all matches of a pattern than the matched text, [finditer()](https://docs.python.org/3/library/re.html#re.finditer) is useful as it provides [match objects](https://docs.python.org/3/library/re.html#match-objects) instead of strings. Continuing with the previous example, if one was a writer who wanted to find all of the adverbs *and their positions* in some text, he or she would use [finditer()](https://docs.python.org/3/library/re.html#re.finditer) in the following manner:

>>>

**>>>** text = "He was carefully disguised but captured quickly by police."

**>>> for** m **in** re.finditer(r"\w+ly", text):

**...**  print('*%02d*-*%02d*: *%s*' % (m.start(), m.end(), m.group(0)))

07-16: carefully

40-47: quickly

## Search and Replace

re.sub(pattern, new\_string\_for\_replacement, string, max=0)

This method replaces all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* provided. And then returns the modified string

Example:

#!/usr/bin/python

import re

phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments

num = re.sub(r'#.\*$', "", phone)

print "Phone Num : ", num

* **re.compile()**

re.**compile**(pattern)

The re.compile( ) function will compile the pattern into pattern objects. After the

compilation the pattern objects will be able to access methods for various operations

like searching,substitutions etc

The sequence

prog = re.compile(pattern)

result = prog.match(string)

is equivalent to

result = re.match(pattern, string)

but using [re.compile()](https://docs.python.org/3/library/re.html#re.compile) and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.

# Matching a string

**Script 1: Write a script to determine if the given substring is present in the string.**

def search\_string():

import re

substring='water'

search1=re.search(substring,'Water water everywhere but not a drop to drink')

if search1:

position=search1.start()

print "matched", substring, "at position", position

else:

print "No match found"

**Script 2: Write a script to determine if the given substring (defined using meta**

**characters) is present in the given string**

def metasearch():

import re

p=re.compile('sing+')

search1=re.search(p,'Some singers sing well')

if search1:

match=search1.group()

index=search1.start()

lindex=search1.end()

print "matched", match, "at index", index ,"ending at" ,lindex

else:

print "No match found"

**CASE STUDY 1:**

>>> s = '100 NORTH MAIN ROAD'

>>> s.replace('ROAD', 'RD.') ①

'100 NORTH MAIN RD.'

>>> s = '100 NORTH BROAD ROAD'

>>> s.replace('ROAD', 'RD.') ②

'100 NORTH BRD. RD.'

>>> s[:-4] + s[-4:].replace('ROAD', 'RD.') ③

'100 NORTH BROAD RD.'

>>> import re ④

>>> re.sub('ROAD$', 'RD.', s) ⑤

'100 NORTH BROAD RD.

Reference for Case Study

<http://www.diveintopython3.net/regular-expressions.html>

|  |  |
| --- | --- |
| ① | My goal is to standardize a street address so that 'ROAD' is always abbreviated as 'RD.'. At first glance, I thought this was simple enough that I could just use the string method replace(). After all, all the data was already uppercase, so case mismatches would not be a problem. And the search string, 'ROAD', was a constant. And in this deceptively simple example, s.replace() does indeed work. |
| ② | Life, unfortunately, is full of counterexamples, and I quickly discovered this one. The problem here is that 'ROAD' appears twice in the address, once as part of the street name 'BROAD'and once as its own word. The replace() method sees these two occurrences and blindly replaces both of them; meanwhile, I see my addresses getting destroyed. |
| ③ | To solve the problem of addresses with more than one 'ROAD' substring, you could resort to something like this: only search and replace 'ROAD' in the last four characters of the address (s[-4:]), and leave the string alone (s[:-4]). But you can see that this is already getting unwieldy. For example, the pattern is dependent on the length of the string you’re replacing. (If you were replacing 'STREET' with 'ST.', you would need to use s[:-6] and s[-6:].replace(...).) Would you like to come back in six months and debug this? I know I wouldn’t. |
| ④ | It’s time to move up to regular expressions. In Python, all functionality related to regular expressions is contained in the re module. |
| ⑤ | Take a look at the first parameter: 'ROAD$'. This is a simple regular expression that matches 'ROAD' only when it occurs at the end of a string. The $ means “end of the string.” (There is a corresponding character, the caret ^, which means “beginning of the string.”) Using the re.sub() function, you search the string s for the regular expression 'ROAD$' and replace it with 'RD.'. This matches the ROAD at the end of the string s, but does *not* match the ROAD that’s part of the word BROAD, because that’s in the middle of s. |

**Case Study 2:**

Continuing with my story of scrubbing addresses, I soon discovered that the previous example, matching 'ROAD' at the end of the address, was not good enough, because not all addresses included a street designation at all. Some addresses simply ended with the street name. I got away with it most of the time, but if the street name was 'BROAD', then the regular expression would match 'ROAD' at the end of the string as part of the word 'BROAD', which is not what I wanted.

>>> s = '100 BROAD'

>>> re.sub('ROAD$', 'RD.', s)

'100 BRD.'

>>> re.sub('\\bROAD$', 'RD.', s) ①

'100 BROAD'

>>> re.sub(r'\bROAD$', 'RD.', s) ②

'100 BROAD'

>>> s = '100 BROAD ROAD APT. 3'

>>> re.sub(r'\bROAD$', 'RD.', s) ③

'100 BROAD ROAD APT. 3'

>>> re.sub(r'\bROAD\b', 'RD.', s) ④

'100 BROAD RD. APT 3'

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
| ① | What I *really* wanted was to match 'ROAD' when it was at the end of the string *and* it was its own word (and not a part of some larger word). To express this in a regular expression, you use \b, which means “a word boundary must occur right here.” In Python, this is complicated by the fact that the '\' character in a string must itself be escaped. This is sometimes referred to as the backslash plague, and it is one reason why regular expressions are easier in Perl than in Python. On the down side, Perl mixes regular expressions with other syntax, so if you have a bug, it may be hard to tell whether it’s a bug in syntax or a bug in your regular expression. |
| ② | To work around the backslash plague, you can use what is called a *raw string*, by prefixing the string with the letter r. This tells Python that nothing in this string should be escaped; '\t'is a tab character, but r'\t' is really the backslash character \ followed by the letter t. I recommend always using raw strings when dealing with regular expressions; otherwise, things get too confusing too quickly (and regular expressions are confusing enough already). |
| ③ | *\*sigh\** Unfortunately, I soon found more cases that contradicted my logic. In this case, the street address contained the word 'ROAD' as a whole word by itself, but it wasn’t at the end, because the address had an apartment number after the street designation. Because 'ROAD' isn’t at the very end of the string, it doesn’t match, so the entire call to re.sub() ends up replacing nothing at all, and you get the original string back, which is not what you want. |
| ④ | To solve this problem, I removed the $ character and added another \b. Now the regular expression reads “match 'ROAD' when it’s a whole word by itself anywhere in the string,” whether at the end, the beginning, or somewhere in the middle. |

<http://www.diveintopython3.net/regular-expressions.html>