re.match()

```
re.match(pattern,
"spamspamspam")
#returns True
```

match returns an object representing the match, if not, it returns None.

Sub()

```
re.sub(pattern, repl, string,
count=0)
str = "My name is David. Hi
David."
pattern = r"David"
newstr = re.sub(pattern, "Amy",
str)
print(newstr)
>>>
My name is Amy. Hi Amy.
>>>
```

This method replaces all occurrences of the pattern in string with repl, substituting all occurrences, unless count provided. This method returns the modified string.

^start &end

```
pattern = r"^gr.y$"
```

The next two metacharacters are ^ and $. These match the start and end of a string, respectively.

[] character classes 3

```
pattern = r"[^A-Z]"
if re.search(pattern, "this is
all quiet"):
   print("Match 1")
if re.search(pattern, "AbCdEf-
G123"):
   print("Match 2")
if re.search(pattern, "THISIS-
ALLSHOUTING"):
   print("Match 3")
##The pattern [^A-Z] excludes
uppercase strings.
Note, that the ^ should be
inside the brackets to invert
the character class.
>>>
Match 1
Match 2
>>>
```

? metacharacter

```
pattern = r"ice(-)?cream"
if re.match(pattern, "ice-cr-
eam"):
   print("Match 1")
if re.match(pattern, "icecre-
am"):
   print("Match 2")
```

The metacharacter ? means "zero or one repetitions".

{} metacharacters

```
pattern = r"9{1,3}$"
if re.match(pattern, "9"):
   print("Match 1")
if re.match(pattern, "999"):
   print("Match 2")
if re.match(pattern, "9999"):
   print("Match 3")
>>>
Match 1
Match 2
>>>
```

Curly braces can be used to represent the number of repetitions between two numbers.
The regex {x,y} means "between x and y repetitions of something".
Hence {0,1} is the same thing as ?.
If the first number is missing, it is taken to be zero. If the second number is missing, it is taken to be infinity.

special sequences

```
pattern = r"(.+) \1"
match = re.match(pattern, "word
word")
if match:
   print ("Match 1")
match = re.match(pattern, "?!
?!")
if match:
   print ("Match 2")
match = re.match(pattern, "abc
cde")
```

special sequences (cont)

```
if match:
   print ("Match 3")
>>>
Match 1
Match 2
>>>
Note, that "(.+) \1" is not the
same as "(.+) (.+)", because \1
refers to the first group's
subexpression, which is the
matched expression itself, and
not the regex pattern.
```

There are various special sequences you can use in regular expressions. They are written as a backslash followed by another character.
One useful special sequence is a backslash and a number between 1 and 99, e.g., \1 or \17. This matches the expression of the group of that number.

search() and findall()

```
if re.search(pattern,
"eggspamsausagespam"):
   print("Match")
else:
   print("No match")

print(re.findall(pattern, "egg-
spamsausagespam"))
>>>
Match
['spam', 'spam']
>>>
```

The function re.search finds a match of a pattern anywhere in the string.
The function re.findall returns a list of all substrings that match a pattern.

. (dot).

```
pattern = r"gr.y"
# this will be grey or gray or
anything else except newline
```

This matches any character, other than a new line.

[] character classes 2

```
pattern = r"[A-Z][A-Z][0-9]"
if re.search(pattern, "LS8"):
   print("Match 1")
if re.search(pattern, "E3"):
   print("Match 2")
#The pattern in the example
above matches strings that
contain two alphabetic uppercase
letters followed by a digit.
>>>
Match 1
>>>
```

Character classes can also match ranges of characters.
The class [a-z] matches any lowercase alphabetic character.
The class [G-P] matches any uppercase character from G to P.
The class [0-9] matches any digit.
Multiple ranges can be included in one class. For example, [A-Za-z] matches a letter of any cases.

+ metacharacter

```
pattern = r"g+"
if re.match(pattern, "g"):
   print("Match 1")
To summarize:
* matches 0 or more occurrences
of the preceding expression.
+ matches 1 or more occurrence
of the preceding expression.
```

The metacharacter + is very similar to *, except it means "one or more repetitions", as opposed to "zero or more repetitions".

Groups in metacharacters ()

```
pattern = r"a(bc)(de)(f(g)h)i"
match = re.match(pattern, "abc-
defghijklmnop")
if match:
   print(match.group())
   print(match.group(0))
   print(match.group(1))
   print(match.group(2))
   print(match.groups())
>>>
abcdefghi
abcdefghi
bc
de
('bc', 'de', 'fgh', 'g')
>>>
```

The content of groups in a match can be accessed using the group function.
A call of group(0) or group() returns the whole match.
A call of group(n), where n is greater than 0, returns the nth group from the left.
The method groups() returns all groups up from 1.

\d \s \w Special sequences

```
pattern = r"(\D+\d)"
match = re.match(pattern, "Hi 999!")
if match:
   print("Match 1")
match = re.match(pattern, "1, 23, 456!")
if match:
   print("Match 2")
match = re.match(pattern, " ! $?")
if match:
    print("Match 3")
>>>
Match 1
>>>
```

More useful special sequences are \d, \s, and \w.
These match digits, whitespace, and word characters respectively.
In ASCII mode they are equivalent to [0-9], [\t\n\r\f\v], and [a-zA-Z0-9_].
In Unicode mode they match certain other characters, as well. For instance, \w matches letters with accents.
Versions of these special sequences with upper case letters - \D, \S, and \W - mean the opposite to the lower-case versions. For instance, \D matches anything that isn't a digit.

Search->>Group, Start,End,Span

```
match = re.search(pattern, "eggspamsausage")
if match:
   print(match.group())
   print(match.start())
   print(match.end())
   print(match.span())
>>>
pam
4
7
(4, 7)
>>>
```

The regex search returns an object with several methods that give details about it.
These methods include group which returns the string matched, start and end which return the start and ending positions of the first match, and span which returns the start and end positions of the first match as a tuple.

[] character classes

```
pattern = r"[aeiou]"
if re.search(pattern, "grey"):
   print("Match 1")
if re.search(pattern, "qwertyuiop"):
   print("Match 2")
if re.search(pattern, "rhythm myths"):
   print("Match 3")
```

[] character classes (cont)

```
#The pattern [aeiou] in the search function matches all strings that contain any one of the characters defined
>>>
Match 1
Match 2
>>>
```

Character classes provide a way to match only one of a specific set of characters.

* metacharacter

```
pattern = r"egg(spam)*"
if re.match(pattern, "egg"):
   print("Match 1")
if re.match(pattern, "eggspamspamegg"):
   print("Match 2")
if re.match(pattern, "spam"):
   print("Match 3")
>>>
match 1
match 2
>>>
The example above matches strings that start with "egg" and follow with zero or more "spam"s.
...............................................
```

The metacharacter * means "zero or more repetitions of the previous thing".

named groups & noncapturing groups

```
pattern = r"(?P<first>abc)
(?:def)(ghi)"
match = re.match(pattern, "abc-
defghi")
if match:
   print(match.group("first"))
   print(match.groups())
>>>
abc
('abc', 'ghi')
>>>
```

Named groups have the format (?P<na-me>...), where name is the name of the group, and ... is the content. They behave exactly the same as normal groups, except they can be accessed by group(name) in addition to its number.
Non-capturing groups have the format (?:...). They are not accessible by the group method, so they can be added to an existing regular expression without breaking the numbering.

| "or" metacharacter

```
pattern = r"gr(a|e)y"
match = re.match(pattern, "gra-
y")
if match:
   print ("Match 1")
match = re.match(pattern, "gre-
y")
if match:
   print ("Match 2")
match = re.match(pattern, "gri-
y")
if match:
```

| "or" metacharacter (cont)

```
    print ("Match 3")
>>>
Match 1
Match 2
>>>
```

Another important metacharacter is |.
This means "or", so red|blue matches either "red" or "blue".

\A \Z \b special sequences

```
pattern = r"\b(cat)\b"
match = re.search(pattern, "The
cat sat!")
if match:
   print ("Match 1")
match = re.search(pattern, "We
s>cat<tered?")
if match:
   print ("Match 2")
match = re.search(pattern, "We
scattered.")
if match:
   print ("Match 3")
>>>
Match 1
Match 2
>>>
"\b(cat)\b" basically matches
the word "cat" surrounded by
word boundaries.
```

Additional special sequences are \A, \Z, and \b.
The sequences \A and \Z match the beginning and end of a string, respectively.
The sequence \b matches the empty string between \w and \W characters, or \w characters and the beginning or end of the string. Informally, it represents the boundary between words.
The sequence \B matches the empty string anywhere else.