Regular Experession in Python

What is regular Expression?

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern in a string. RegEx can be used to check if a string contains the specified search pattern. Regular Expression is independent of any programming language, and you can find general rules of regular expression on [this link. (https://en.wikipedia.org/wiki/Regular_expression)](https://en.wikipedia.org/wiki/Regular_expression)

> Regular expressions are used in search engines, search and replace dialogs of word processors and text editors, in text processing utilities such as sed and AWK and in lexical analysis. Many programming languages provide regex capabilities either built-in or via libraries.

Python module re - provides regular expression matching operations similar to those found in Perl. [Here is original documentation of re module. (https://docs.python.org/3/library/re.html)](https://docs.python.org/3/library/re.html) Also, gentle introduction to usage of Regular expression in python is available [Here. (https://docs.python.org/3/howto/regex.html#regex-howto)](https://docs.python.org/3/howto/regex.html#regex-howto)

- Refereces

https://github.com/python/cpython/blob/3.8/Lib/re.py (https://github.com/python/cpython/blob/3.8/Lib/re.py)

https://docs.python.org/3/library/re.html (https://docs.python.org/3/library/re.html)

In [1]:

```
import re
```

re module exports the following functions:

```
match     Match a regular expression pattern to the beginning of a string.
fullmatch Match a regular expression pattern to all of a string.
search    Search a string for the presence of a pattern.
sub       Substitute occurrences of a pattern found in a string.
subn      Same as sub, but also return the number of substitutions made.
split     Split a string by the occurrences of a pattern.
findall   Find all occurrences of a pattern in a string.
finditer  Return an iterator yielding a Match object for each match.
compile   Compile a pattern into a Pattern object.
purge     Clear the regular expression cache.
escape    Backslash all non-alphanumerics in a string.
```

In [2]:

```
'''
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.
Return None if the string does not match the pattern;

Note that even in MULTILINE mode, re.match() will only match at the
beginning of the string and not at the beginning of each line.
'''

print(re.match("if", "if else if If else ok ok"))
```

```
<re.Match object; span=(0, 2), match='if'>
```

In [3]:

```
'''
re.fullmatch(pattern, string, flags=0)

If the whole string matches the regular expression pattern, return a corresponding match ob
Return None if the string does not match the pattern;

'''
print(re.fullmatch("If", "If"))
```

```
<re.Match object; span=(0, 2), match='If'>
```

In [4]:

```
'''
re.search(pattern, string, flags=0)

Scan through string looking for the first location
where the regular expression pattern produces a match, and
return a corresponding match object.
Return None if no position in the string matches the pattern;

'''

print(re.search("if", "If if if"))
```

```
<re.Match object; span=(3, 5), match='if'>
```

In [5]:

```
'''
re.sub(pattern, repl, string, count=0, flags=0)

  Return the string obtained by replacing pattern in string by the repl.
  If the pattern isn't found, string is returned unchanged.
  repl can be a string or a function; // IMP
  if it is a string, any backslash escapes in it are processed.
  That is, \n is converted to a single newline character.
  Unknown escapes of ASCII letters are reserved for future use and treated as errors.
  Other unknown escapes such as \& are left alone.
  Backreferences, such as \6, are replaced with the substring matched by group 6 in the pat
'''
def myfunc():

  return "Some String"

print(re.sub("if", myfunc() , "if else if IFl", 1))
```

```
Some String else if IFl
```

In [6]:

```
'''
re.subn(pattern, repl, string, count=0, flags=0)

Perform the same operation as sub(), but return a tuple (new_string, number_of_subs_made).
'''
subns = re.subn("if","my" , "if else if IFl")

print(subns)
print(subns[0])
```

```
('my else my IFl', 2)
my else my IFl
```

In [7]:

```
'''
re.split(pattern, string, maxsplit=0, flags=0)

Split string by the occurrences of pattern.
If capturing parentheses are used in pattern,
then the text of all groups in the pattern are also returned as part of the resulting list.
If maxsplit is nonzero, at most maxsplit splits occur,
and the remainder of the string is returned as the final element of the list.
'''

print(re.split("f","if else if IFl", 1))
```

```
['i', ' else if IFl']
```

In [8]:

```
'''
re.findall(pattern, string, flags=0)

Return all non-overlapping matches of pattern in string, as a list of strings.
The string is scanned left-to-right, and matches are returned in the order found.
If one or more groups are present in the pattern, return a list of groups;
this will be a list of tuples if the pattern has more than one group.
Empty matches are included in the result.
'''

print(re.findall("se","if else if and else if ok"))
```

```
['se', 'se']
```

In [9]:

```
'''
re.finditer(pattern, string, flags=0)

Return an iterator yielding match objects over all non-overlapping matches
for the RE pattern in string.
The string is scanned left-to-right, and matches are returned in the order found.
'''
print(re.finditer("if","if else if and else if ok"))

rex = re.finditer("if","if else if and else if ok")
for i in rex:
  print(i)
```

```
<callable_iterator object at 0x00000182B775B160>
<re.Match object; span=(0, 2), match='if'>
<re.Match object; span=(8, 10), match='if'>
<re.Match object; span=(20, 22), match='if'>
```

In [10]:

```
'''
re.compile(pattern, flags=0)

Compile a regular expression pattern into a regular expression object,
which can be used for matching using its match(), search() and other methods.

The expression's behaviour can be modified by specifying a flags value.
Flag Values can be any of the re flag variables, combined using bitwise OR (the | operator)

Note: Using 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.
'''
string = "a is good at a"
pattern = "^a[a-z ]*a$" # Start with a, followed by any small character or space and  zero
                        # and should end with small a"
patt = re.compile(pattern)
result = patt.match(string)
print(result)
result = re.match(pattern, string)
print(result)
```

```
<re.Match object; span=(0, 14), match='a is good at a'>
<re.Match object; span=(0, 14), match='a is good at a'>
```

In [11]:

```
'''
re.purge()
  Clear the regular expression cache.
'''
```

Out[11]:

```
'\nre.purge()\n  Clear the regular expression cache.\n'
```

In [12]:

```
'''
re.escape(pattern)
Escape special characters in pattern.
This is useful if you want to match an arbitrary literal string
that may have regular expression metacharacters in it.
'''
print(re.escape("h.(h)"))
```

```
h\.\(h\)
```

Match Objects

match object exports the following functions:

```
match     Match a regular expression pattern to the beginning of a string.
fullmatch Match a regular expression pattern to all of a string.
search    Search a string for the presence of a pattern.
finditer  Return an iterator yielding a Match object for each match.
```

In [13]:

```
pattern = "if"
string = "if if else if and ok"
match = re.search(pattern, string)
print(match)
```

```
<re.Match object; span=(0, 2), match='if'>
```

In [14]:

```
'''
Match.expand(template)

Return the string obtained by doing backslash substitution
on the template string, as done by the sub() method.
Escapes such as \n are converted to the appropriate characters,
 and numeric backreferences (\1, \2) and named backreferences (\g<1>, \g<name>)
 are replaced by the contents of the corresponding group.
'''

match = re.search("(\d\d\d\d) (\d\d\d\d)", "in the year 1999 2000")
print(match)
print(match.expand(r"Year: \1"))    # Year: 1999
```

```
<re.Match object; span=(12, 21), match='1999 2000'>
Year: 1999
```

groups(), group(), groupdict():

```
group()     Match a regular expression pattern to the beginning of a string.
groups()    Match a regular expression pattern to all of a string.
groupdict() Search a string for the presence of a pattern.
```

In [15]:

```
'''
Match.group([group1, ...])

Returns one or more subgroups of the match.
If there is a single argument, the result is a single string;
if there are multiple arguments, the result is a tuple with one item per argument.
Without arguments, group1 defaults to zero (the whole match is returned).
If a groupN argument is zero, the corresponding return value is the entire matching string;
if it is in the inclusive range [1..99], it is the string matching the corresponding parent
If a group number is negative or larger than the number of groups defined in the pattern,
an IndexError exception is raised.
If a group is contained in a part of the pattern that did not match,
the corresponding result is None.
If a group is contained in a part of the pattern that matched multiple times, the last matc
'''

myText = 'some 11 22 44 list'
matchObj = re.search(".+ (\d+) (\d+) (\d+) .+",myText)

print(matchObj.group())     # 'some 11 22 33 list'
print(matchObj.group(0))    # 'some 11 22 33 list'
print(matchObj.group(1))    # '11'
print(matchObj.group(2))    # '22'
print(matchObj.group(3))    # '33'
print(matchObj.group(1,3))  # ('11', '22')
print(matchObj.group(2,1,1))# ('22', '11', '11')
print(matchObj.group(0,1))  # ('some 11 22 33 list', '11')
```

```
some 11 22 44 list
some 11 22 44 list
11
22
44
('11', '44')
('22', '11', '11')
('some 11 22 44 list', '11')
```

In [16]:

```
'''
Match.groups(default=None)

Return a tuple containing all the subgroups of the match,
from 1 up to however many groups are in the pattern.
The default argument is used for groups that did not participate in the match;
it defaults to None.
'''
myText = 'some 11 22 33 list'
matchObj = re.search(".+ (\d+) (\d+) (\d+) .+",myText)

print(matchObj.groups()) # ('11', '22', '33')
```

```
('11', '22', '33')
```

In [17]:

```
'''
Match.groupdict(default=None)

Return a dictionary containing all the named subgroups of the match,
keyed by the subgroup name.
The default argument is used for groups that did not participate in the match;
it defaults to None.
'''

myText = 'some 11 22 33 list'
patObj = re.compile('.+ (?P<first>\d+) (?P<second>\d+) (?P<third>\d+).+')
matchObj=patObj.search(myText)
print(matchObj.groupdict())
```

```
{'first': '11', 'second': '22', 'third': '33'}
```

start(…), end(…), span(…):

start(n)	return the index where the n captured group begins.
end(n)	return the index where the n captured group ends
span(n)	return a tuple, with start and end position of the nth captured group

In [18]:

```
'''
Match.start([group])
Match.end([group])

Return the indices of the start and end of the substring matched by group;
group defaults to zero (meaning the whole matched substring).
Return -1 if group exists but did not contribute to the match.
'''
myText = 'some 11 22 33 list'
matchObj = re.search(".+(\d+) (\d+) (\d+).+",myText)
print(matchObj.start(1))
print(matchObj.end(1))
```

```
6
7
```

In [19]:

```
'''
Match.span([group])

For a match m, return the 2-tuple (m.start(group), m.end(group)).
Note that if group did not contribute to the match,
this is (-1, -1).
group defaults to zero, the entire match.
'''
myText = 'some 11 22 33 list'
matchObj = re.search(".+(\d+) (\d+) (\d+).+",myText)
print(matchObj)
print(matchObj.span(1))
```

```
<re.Match object; span=(0, 18), match='some 11 22 33 list'>
(6, 7)
```

Match Object Attributes

```
Match.string      The string passed to match() or search().
Match.pos         This is the index into the string at which the RE engine started
looking for a match.
Match.endpos      This is the index into the string beyond which the RE engine wil
l not go.
Match.lastindex   The integer index of the last matched capturing group, or None i
f no group was matched at all.
Match.lastgroup   The name of the last matched capturing group, or None if the gro
up didn’t have a name, or if no group was matched at all.
Match.re          The regular expression object whose match() or search() method p
roduced this match instance.
```

Compiled Regular Expression Objects

```
Pattern.search(string[, pos[, endpos]])
Pattern.match(string[, pos[, endpos]])
Pattern.fullmatch(string[, pos[, endpos]])
Pattern.split(string, maxsplit=0)
Pattern.findall(string[, pos[, endpos]])
Pattern.finditer(string[, pos[, endpos]])
Pattern.sub(repl, string, count=0)
Pattern.subn(repl, string, count=0)
```

Properties

```
Pattern.flags
Pattern.groups
Pattern.groupindex
Pattern.pattern
```

In [20]:

```
string = 'some 33 22 33 list'
pattern = re.compile("33")
match = pattern.search(string, 8)
print(match)
```

```
<re.Match object; span=(11, 13), match='33'>
```

re Flags

re.A | re.ASCII

```
Make \w, \W, \b, \B, \d, \D, \s and \S perform
ASCII-only matching instead of full Unicode matching.
This is only meaningful for Unicode patterns,
and is ignored for byte patterns.
```

re.DEBUG

```
Display debug information about compiled expression.
No corresponding inline flag.
```

re.I | re.IGNORECASE

```
Perform case-insensitive matching;
expressions like [A-Z] will also match lowercase letters.
```

re.L | re.LOCALE

```
Make \w, \W, \b, \B and case-insensitive matching dependent on the current locale.
This flag can be used only with bytes patterns.
The use of this flag is discouraged as the locale mechanism is very unreliable, it
only handles one “culture” at a time, and it only works with 8-bit locales.
```

re.M | re.MULTILINE

```
When specified, the pattern character '^' matches at the beginning of the string a
nd
at the beginning of each line (immediately following each newline);
and the pattern character '$' matches at the end of the string and
    at the end of each line (immediately preceding each newline).
    By default, '^' matches only at the beginning of the string,
    and '$' only at the end of the string and immediately before the newline (if a
ny) at the end of the string. Corresponds to the inline flag (?m).
```

re.S | re.DOTALL

```
Make the '.' special character match any character at all, including a newline;
without this flag, '.' will match anything except a newline.
```

re.X | re.VERBOSE

This flag allows you to write regular expressions that look nicer and are more readable
by allowing you to visually separate logical sections of the pattern and add comments.

```
a = re.compile("""\d +  # the integral part
                  \.    # the decimal point
                  \d *  # some fractional digits""", (re.X,re.M))
b = re.compile(r"\d+\.\d*")
```