Python - Advanced - Regular Expressions

September 13, 2017

1 Regular Expression

Regular expressions (or regex) are used for pattern matching. Almost all major languages support the regexs. Python has **re** module to deal with regular expressions.

```
In [1]: import re

In [2]: print dir(re)

['DEBUG', 'DOTALL', 'I', 'IGNORECASE', 'L', 'LOCALE', 'M', 'MULTILINE', 'S', 'Scanner', 'T', '
```

1.1 Creating regex(pattern) Object

```
In [3]: regObject = re.compile("python")

In [5]: regObject

Out[5]: re.compile(r'python')

In [6]: type(regObject)

Out[6]: _sre.SRE_Pattern

In [7]: print dir(regObject)

['__class__', '__copy__', '__deepcopy__', '__delattr__', '__doc__', '__format__', '__getattribu
```

1.2 re.match()

re.match(string[, pos[, endpos]]) - Matches zero or more characaters at the beginning of the string.

```
In [8]: regObject.match("python Programming is Good")

Out[8]: <_sre.SRE_Match at 0x37ccad8>
```

The matched result is stored in re type object

```
In [9]: matchedResult = regObject.match("python Programming is Good")

In [10]: matchedResult

Out[10]: <_sre.SRE_Match at 0x37ccb10>

In [11]: matchedResult.group()

Out[11]: 'python'

In [12]: matchedResult = regObject.match("Programming python is Good")

In [13]: matchedResult

In [14]: print matchedResult # Because it tries to loacte at starting

None


In [15]: matchedResult.group()

        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-15-39c57b0577ad> in <module>()
    ----> 1 matchedResult.group()


        AttributeError: 'NoneType' object has no attribute 'group'
```

NOTE: when there isn't any match, matchedResult Object doesn't hav the *group()* attribute.

```
In [16]: print regObject.match("Programming python is Good").group()


        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-16-3d10c4e92b98> in <module>()
    ----> 1 print regObject.match("Programming python is Good").group()


        AttributeError: 'NoneType' object has no attribute 'group'


In [17]: print regObject.match(" python Programming is Good")
                # string is starting with  a white-space
```

```
None
```

```
In [18]: print regObject.match("Python Programming is Good")
                # capital 'P' encountered
```

```
None
```

```
In [19]: print regObject.match('pythoN')
                # case-sensitivity
```

```
None
```

```
In [20]: print regObject.match('pythoN'.lower())
```

```
<_sre.SRE_Match object at 0x038971A8>
```

```
In [21]: print regObject.match('pythoN'.lower()).group()
```

```
python
```

```
In [22]: #!/usr/bin/python
         import re

         checkString = 'python'
         compiledCheckString = re.compile(checkString)
           # re object was be created
         targetString = 'python programming'
         matchs = compiledCheckString.match(targetString)
           # match() will be initiated on the re object
         print matchs
         if matchs:
             print "Now lets print the matches word"
             print matchs.group()
```

```
<_sre.SRE_Match object at 0x03897288>
Now lets print the matches word
python
```

```
In [23]: #!/usr/bin/python
         import re

         checkString = 'python'
         compiledCheckString = re.compile(checkString)
             # re object was be created
```

```
targetString = 'Python'
matchs = compiledCheckString.match(targetString)
    # match() will be initiated on the  re object
print matchs
if matchs:
    print "Now lets print the matches word"
    print matchs.group()
else:
    print "No matches found"
```

```
None
No matches found
```

Question: What is the difference between re.search() and re.match()?

1.3 re.search()

re.search(string[, pos[, endpos]]) - Matches zero or more characters anywhere in the string.

```
In [24]: regObject = re.compile("python")
```

For a pattern present at the starting of the string,

```
In [25]: regObject.search("python programming")

Out[25]: <_sre.SRE_Match at 0x3897288>

In [26]: regObject.match("python programming")

Out[26]: <_sre.SRE_Match at 0x38973a0>
```

For a pattern present somewhere except at starting of the string,

```
In [27]: regObject.search("programming python is good")

Out[27]: <_sre.SRE_Match at 0x3897410>

In [28]: print regObject.match("programming python is good")

None


In [29]: regObject.search("programming is good in python")

Out[29]: <_sre.SRE_Match at 0x3897528>

In [30]: regObject = re.compile("python is")

         print regObject.match("programming python is good")
         print regObject.search("programming python is good")
```

```
None
<_sre.SRE_Match object at 0x038975D0>


In [31]: print regObject.search("is programming python in good ?")

None
```

1.4 Special regex Characters

1.4.1 ^(caret)

- Matches the start of the string.

```
In [32]: myString = "This is python class"
         regObject = re.compile("^This")

In [33]: matchResult = regObject.match(myString)

In [34]: print matchResult.group()

This
```

In Other way,

```
In [35]: re.match('^This', myString)

Out[35]: <_sre.SRE_Match at 0x3897838>

In [36]: re.match('^This', myString).group()

Out[36]: 'This'

In [37]: re.match('^This', "This is python class").group()

Out[37]: 'This'

In [38]: re.search('^This',"This is python class").group()

Out[38]: 'This'

In [39]: print re.search('^This', "Yes!, This is python class")

None


In [40]: print re.match('^This', "Yes!, This is python class")
             # It isn't meaningful

None
```

1.4.2 $

- Matches the end of the strig, or just before the newlie at the end of the string.

```
In [41]: myString = 'foo foobar'
         regObject= re.compile('foobar$')

In [44]: print regObject.match(myString)

None


In [45]: print regObject.search(myString)

<_sre.SRE_Match object at 0x03897BF0>


In [46]: print regObject.search(myString).group()

foobar
```

In Other way,

```
In [48]: print re.match('foobar$',myString)

None


In [49]: print re.search('foobar$',myString)

<_sre.SRE_Match object at 0x03897CD0>


In [50]: print re.search('foobar$',myString).group()

foobar
```

1.4.3 .(dot)

- Matches **any character**, except the newline.
- If DOTALL flag is enabled, it matches any character including newline.

```
In [51]: #!usr/bin/python
         import re

         string = 'This'
         result = re.search('....', string)

         if result:
             print result.group()
```

```
This
```

```
In [56]: print re.search('......', "batman").group()
```

```
batman
```

```
In [57]: print re.search('....', "batman").group()
```

```
batm
```

Question: What is the result of re.search(", "batman").group()?

```
In [58]: re.search('', "batman").group()
```

```
Out[58]: ''
```

```
In [59]: print re.search('....$', "batman").group()
```

```
tman
```

```
In [60]: print re.search('^....', "batman").group()
```

```
batm
```

```
In [61]: print re.search('^....$', "batman").group()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)

<ipython-input-61-169edd47f0ce> in <module>()
----> 1 print re.search('^...$', "batman").group()


AttributeError: 'NoneType' object has no attribute 'group'
```

```
In [62]: print re.search('^......$', "batman").group()
```

```
batman
```

1.4.4 * (astrick)

- causes the RE to match **0 or more** repetitions of the preceeding RE. `Ex: 'ab*' - matches for 'a', 'ab', and 'a' followed by any number of 'b''s ab, abb, abbbb ...`

```
In [63]: print string

This


In [64]: re.search('.*',string).group()

Out[64]: 'This'

In [66]: re.search('.*', "I want to become a backend developer").group()

Out[66]: 'I want to become a backend developer'

In [71]: re.search('. *', "I want to become a backend developer").group()

Out[71]: 'I '

In [67]: re.search('.w*', "I want to become a backend developer").group()

Out[67]: 'I'

In [69]: re.search('.b*', "I want to become a backend developer").group()

Out[69]: 'I'

In [72]: re.search('.Q*', "I want to become a backend developer").group()
                # 'Q' isn't in the search string

Out[72]: 'I'

In [73]: re.search('w*', "I want to become a backend developer").group()

Out[73]: ''

In [74]: re.search('.*', '').group() # trying to find in null string
                 # resulted in white-space, not NONE

Out[74]: ''
```

1.4.5 + (plus)

- causes the RE to match **1 or more** repetitions of the preceeding RE.

Ex: ab+ -> 'ab', 'abb', 'abbb', ...

```
In [75]: string = 'abbba'

         re.match('ab+a',string).group()

Out[75]: 'abbba'

In [76]: print re.match('ab+a','aa')

None


In [77]: print re.match('ab*a','aa123').group()
             # # * tries to find 0 or more occurrences of 'b'

aa


In [78]: print re.match('ab*c','aa123').group()


        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-78-b324747346bc> in <module>()
    ----> 1 print re.match('ab*c','aa123').group()


        AttributeError: 'NoneType' object has no attribute 'group'


In [79]: print re.match('ab*c','aca123').group()

ac


In [80]: print re.match('ab*.*','aca123').group()

aca123


In [81]: re.search('ab+a','abbba').group()

Out[81]: 'abbba'

In [82]: print re.search('ab+a','aa')

None
```

1.4.6 ? (question mark)

- causes the RE to match **0 or 1 time ONLY** of the preceeding RE

Ex: ab? --> 'a', 'ab'

```
In [83]: string = 'hello'

In [84]: print re.match(r'hello?', string).group()

hello


In [85]: print re.match(r'hello?', 'hell').group()

hell


In [86]: print re.match(r'hello?', 'hel')

None


In [87]: print re.match(r'hell?o?', 'hel').group()

hel


In [88]: print re.match('hell?o', 'helll')
                 # 'l' is repeating 3 times

None


In [89]: print re.match('hell?o', 'helllo')
                 # 'l' is repeating 3 times

None


In [90]: print re.match('hell?o', 'helo').group()

helo
```

Consolidation:

```
*  - 0 or more
+  - 1 or more
?  - 0 or 1
```

1.5 Gready Search Patterns

__*?, +?, ?? __- GREEDY SEARCH Patterns

```
In [91]: string = '<H1>title</H1>'

In [92]: print re.match(r'<.*>', string).group()

<H1>title</H1>


In [93]: print re.match(r'<H*?>', string).group()


         ---------------------------------------------------------------------------

         AttributeError                            Traceback (most recent call last)

         <ipython-input-93-ad2e5d60ff7e> in <module>()
     ----> 1 print re.match(r'<H*?>', string).group()


         AttributeError: 'NoneType' object has no attribute 'group'


In [94]: print re.match(r'<H1*?>', string).group()

<H1>


In [95]: print re.match(r'<.*?>', string).group()

<H1>
```

1.5.1 {m}

- specifies the exactly **m copies** of previous RE

Ex: **a{6}** -- it maches six 'a' characters

```
In [97]: string = 'aaashique'

In [98]: re.match('a{3}shique', string).group()

Out[98]: 'aaashique'

In [100]: print re.match('a{3}shique', 'aashique')

None
```

```
In [104]: re.match('aa{2}shique', 'aaashique').group()
Out[104]: 'aaashique'
In [102]: print re.match('a{3}shique', 'aaaaaashique')
None

In [2]: print re.search('a{3}shique', 'aaaaaashique').group()
aaashique

In [103]: re.match('aaaa{3}shique', 'aaaaaashique').group()
Out[103]: 'aaaaaashique'
```

1.5.2 {m,n}

- causes the resulting RE to match **_from m to n repititions** of the preceding RE

  Ex: **a{3,5}** will match from 3 to 5 'a' characters

```
In [105]: string = 'aaashique'
In [106]: print re.match('a{2,3}shique', string).group()
aaashique

In [107]: print re.match('a{2,3}shique',  'aashique').group()
aashique

In [108]: print re.match('aa{2,3}shique',  'aaaashique').group()
aaaashique
```

1.5.3 {m,n}?

- combined regex pattern

```
In [110]: re.match('a{1,2}?shique',  'aashique').group()
Out[110]: 'aashique'
In [111]: re.match('a{2,3}','aaaaaa').group()
Out[111]: 'aaa'
In [112]: re.search('a{2,3}',string).group()
Out[112]: 'aaa'
In [113]: re.search('a{2,3}?',string).group()
                  # takes 2 occurrences, due to the presence of '?'
Out[113]: 'aa'
```

1.5.4

- Either escapes the special characters (permittin you to match characters like '*', '?') or used to signal a special sequence.

```
In [114]: string = '<H*>test<H*>'

In [115]: print re.match('<H*>',string)

None


In [116]: re.match('<H\*>',string).group()

Out[116]: '<H*>'

In [117]: string = '<H?>test<H?>' # observe, '?' is regex operator

In [118]: re.match('<H\?>',string).group()

Out[118]: '<H?>'
```

1.5.5 []

- used to indicate a set of characters.
- regular expression characters will lose their significance, within the [] (square) braces

Ex:

```
[mnk] - will match the characters 'm', 'n' and 'k'

[a-z] - will match all characters from 'a' to 'z'

[A-Z] - will match all characters from 'A' to 'Z'

[0-9] - will match all characters from 0 to 9

[a-m] - will match all characters from 'a' to 'm'

In [125]: re.match('h[eE]llo','hello').group()

Out[125]: 'hello'

In [126]: re.match('h[eE]llo','hEllo').group()

Out[126]: 'hEllo'

In [121]: print re.match('h[eE]llo','heEllo')

None
```

```
In [122]: print re.match('h[eE]llo','heello')

None


In [127]: re.match('h[eE]*llo','heello').group()

Out[127]: 'heello'

In [124]: re.match('[a-z].*','hello').group()

Out[124]: 'hello'

In [128]: re.match('[a-z].*','hello123').group()

Out[128]: 'hello123'

In [129]: re.match('[a-z]','hello123').group()

Out[129]: 'h'

In [130]: print re.search('[a-z]$','hello123')

None


In [131]: re.search('[0-9]$','hello123').group()

Out[131]: '3'
```

Note: Special characters lose their special meaning inside sets.

To match a literal ']' inside a set, precede it with a backslash, or place it at the beginning of the set.

For example, both [()] and [] () {[]} will match a parenthesis.

```
In [132]: string = '<h*>test<h*>'

In [133]: re.match('<h[*]>',string).group()    # esacaping *

Out[133]: '<h*>'

In [134]: re.match('<h\*>',string).group()     # esacaping *

Out[134]: '<h*>'
```

Question: Write a regular expression, to match all email IDs?

Assignment: Identify the email IDs in the resume in the following paragraph.

```
Welcome to RegExr v2.1 by gskinner.com, proudly hosted by Media Temple!

Edit the Expression & Text to see matches. Roll over matches or the expression for details. Unc
python@programm.com
Sample text for testing:
abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789 _+-.,!@#$%^&*();\/|<>"'python@gmail.com
12345 -98.7 3.141 .6180 9,000 +42
555.123.4567    +1-(800)-555-2468
foo@demo.net    bar.ba@test.co.uk
www.demo.com    http://foo.co.uk/
http://regexr.com/foo.html?q=bar
https://mediatemple.net
mediatepmple@outlook.com
mubeen.tom@hacker.com
1%453&harini_new@in.com
```

1.6 re.IGNORECASE

- to ignore the case (upper and lower), during the match/search.

```
In [137]: regObject = re.compile('python', re.IGNORECASE)
                  #compiling of regex object lets us to reuse it.

          result = regObject.search('PYTHON')
          print result
          if result:
              print result.group()
```

```
<_sre.SRE_Match object at 0x038AC3A0>
PYTHON
```

```
In [138]: reg = re.compile('python', re.I)
                  # Both re.I and re.IGNORECASE work in same way

          result = reg.search('PyThOn')
          print result
          if result:
              print result.group()
```

```
<_sre.SRE_Match object at 0x038AC1A8>
PyThOn
```

```
In [139]: print re.search('python', 'PyThOn', re.I).group()
                       # Alternative method
```

```
PyThOn
```

1.7 re.DOTALL

- special character **match any character** at all, including a **newline**.

```
In [140]: string = 'Today is Friday.\n Tomarrow is morning'
          print string
```

```
Today is Friday.
 Tomarrow is morning
```

```
In [141]: reg = re.compile('.*')

          print reg.search(string).group()
                  # Observe that only first line is matched
```

```
Today is Friday.
```

```
In [142]: reg = re.compile('.*', re.DOTALL)

          print reg.search(string).group()
                  # Now, all the lines will be matched
```

```
Today is Friday.
 Tomarrow is morning
```

```
In [143]: print re.search('.*', string, re.DOTALL).group()
                  # ALTERNATIVELY
```

```
Today is Friday.
 Tomarrow is morning
```

1.8 Grouping

```
\w  - presence of Alphabet
\W  - absence of Alphabet
\d  - presence of digit
\D  - absence of digit
\s  - presence of White-space
\S  - absence of white-space
```

```
In [144]: re.search('\w', 'udhay prakash').group()
```

```
Out[144]: 'u'
```

```
In [145]: re.search('\w*', 'udhay prakash').group()
```

```
Out[145]: 'udhay'
```

```
In [146]: re.search('(\w)', 'udhay prakash').group()

Out[146]: 'u'

In [147]: re.search('(\w*)', 'udhay prakash').group()

Out[147]: 'udhay'

In [148]: re.search('(\w*)', 'udhay prakash').group(0)

Out[148]: 'udhay'
```

NOTE: .group(0) is same as .group()

```
In [150]: re.search('(\w*) (\w*)', 'udhay prakash').group()

Out[150]: 'udhay prakash'

In [151]: re.search('(\w*) (\w*)', 'udhay prakash').group(0)

Out[151]: 'udhay prakash'

In [152]: re.search('(\w*) (\w*)', 'udhay prakash').group(1)

Out[152]: 'udhay'

In [153]: re.search('(\w*) (\w*)', 'udhay prakash').group(2)

Out[153]: 'prakash'

In [154]: re.search('(\w*)(\W)(\w*)', 'udhay prakash').group()

Out[154]: 'udhay prakash'

In [155]: re.search('(\w*)(\W)(\w*)', 'udhay prakash').group(3)

Out[155]: 'prakash'

In [156]: re.search('(\w*)(\s)(\w*)', 'udhay prakash').group()

Out[156]: 'udhay prakash'
```

1.9 Perl based grouping pattern

1.9.1 (?P<name>)

```
In [157]: m = re.match(r"(?P<first>\w+) (?P<last>\w+)", "Udhay Prakash")

In [158]: m.group()

Out[158]: 'Udhay Prakash'

In [159]: m.group(0)
```

```
Out[159]: 'Udhay Prakash'

In [160]: m.group(1)

Out[160]: 'Udhay'

In [163]: m.group('first')

Out[163]: 'Udhay'

In [161]: m.group(2)

Out[161]: 'Prakash'

In [164]: m.group('last')

Out[164]: 'Prakash'

In [165]: print last

        ---------------------------------------------------------------------------
        NameError                                 Traceback (most recent call last)

        <ipython-input-165-2366e70fb8e9> in <module>()
    ----> 1 print last


        NameError: name 'last' is not defined
```

NOTE: Observe that those identifiers can't be used outside.

```
In [166]: re.match(r'(..)+', 'alb2cs').group()

Out[166]: 'alb2cs'
```

NOTE: If a group matches multiple times, only the last match is accessible.

```
In [167]: re.match(r'(..)+', 'alb2cs').group(0)

Out[167]: 'alb2cs'

In [168]: re.match(r'(..)+', 'alb2cs').group(1)

Out[168]: 'cs'

In [169]: re.match(r'(..)+', 'alb2cs').group(2)
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

<ipython-input-169-78100e216102> in <module>()
----> 1 re.match(r'(..)+', 'alb2cs').group(2)


IndexError: no such group
```

Assignment: Try replacing match with search in the below expression, and reevaluate:

```
re.match(r'(..)+', 'alb2cs').group(1)
```

1.9.2 groups([defaullt])

```
In [170]: re.match(r'(..)+', 'batman').groups()

Out[170]: ('an',)
```

Question: what is the difference between .group() and .groups()?

```
In [171]: re.match(r'(..)+', 'batman').group()

Out[171]: 'batman'

In [172]: re.match(r'(\d+)\.(\d+)', '3.45678').groups()

Out[172]: ('3', '45678')

In [174]: re.match(r'(\d+)\.(\d+)', '345678').group()


---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-174-097539aec35b> in <module>()
----> 1 re.match(r'(\d+)\.(\d+)', '345678').group()


AttributeError: 'NoneType' object has no attribute 'group'


In [175]: re.match(r'(\d+)\.?(\d+)', '345678').groups()

Out[175]: ('34567', '8')
```

```
In [176]: re.match(r'(\d+)\.?(\d+)', '1947').groups()
Out[176]: ('194', '7')
In [177]: re.match(r'(\d+)\.?(\d+)', '96').groups()
Out[177]: ('9', '6')
In [178]: re.match(r'(\d+)\.?(\d+)', '96').groups(0)
Out[178]: ('9', '6')
In [179]: re.match(r'(\d+)\.?(\d+)', '96').groups(1)
Out[179]: ('9', '6')
In [180]: re.match(r'(\d+)\.?(\d+)?', '96').groups()
Out[180]: ('96', None)
In [181]: re.match(r'(\d+)\.?(\d+)?', '96').groups(0)
Out[181]: ('96', 0)
In [182]: re.match(r'(\d+)\.?(\d+)?', '96').groups(1)
Out[182]: ('96', 1)
In [183]: re.match(r'(\d+)\.?(\d+)?', '96').groups(35)
Out[183]: ('96', 35)
```

1.9.3 groupdict([default])

```
In [184]: re.match(r'(?P<frst>\w+) (?P<lst>\w+)','Mickel John').groupdict()
Out[184]: {'frst': 'Mickel', 'lst': 'John'}
```

1.9.4 start([group])

- Returns the stating position of the match.

```
In [187]: email = r'myEmail@yahoomail.com'
In [194]: m = re.search('yahoomail', email)
          if m:
              print "domain name is ", m.group()
domain name is  yahoomail

In [190]: m.start()
Out[190]: 8
In [192]: email[:m.start()]
Out[192]: 'myEmail@'
```

1.9.5 end([group])

```
In [191]: m.end()

Out[191]: 17

In [193]: email[m.start():m.end()]

Out[193]: 'yahoomail'
```

1.9.6 span([group])

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

```
In [195]: m.span()

Out[195]: (8, 17)

In [197]: t = m.span();
          print "The domain name is ", email[t[0]: t[1]]

The domain name is  yahoomail
```

1.10 re.findall()

- Used to result in all the occurrences of given pattern.

```
In [198]: re.findall(r'\d', '23456778')

Out[198]: ['2', '3', '4', '5', '6', '7', '7', '8']

In [200]: re.findall(r'\d', '23rd success')

Out[200]: ['2', '3']

In [201]: re.findall(r'\w', '23rd sucess')

Out[201]: ['2', '3', 'r', 'd', 's', 'u', 'c', 'e', 's', 's']

In [202]: re.findall(r'(is)', 'This is good day')

Out[202]: ['is', 'is']
```

\s is used to match white-spaces

```
In [203]: re.findall(r'\s', 'This is good day')

Out[203]: [' ', ' ', ' ']

In [204]: re.findall(r'(\sis)', 'This is good day')

Out[204]: [' is']
```

1.10.1 re.finditer()

- Used to result in all the occurrences of given pattern;
- But, it returns an iterator.

```
In [206]: re.finditer(r'\w', 'this is good day')

Out[206]: <callable-iterator at 0x38f55b0>

In [207]: list(re.finditer(r'\w', 'this is good day'))

Out[207]: [<_sre.SRE_Match at 0x38e4598>,
           <_sre.SRE_Match at 0x38e4608>,
           <_sre.SRE_Match at 0x38e4640>,
           <_sre.SRE_Match at 0x38e4678>,
           <_sre.SRE_Match at 0x38e46b0>,
           <_sre.SRE_Match at 0x38e46e8>,
           <_sre.SRE_Match at 0x38e4720>,
           <_sre.SRE_Match at 0x38e4758>,
           <_sre.SRE_Match at 0x38e4790>,
           <_sre.SRE_Match at 0x38e47c8>,
           <_sre.SRE_Match at 0x38e4800>,
           <_sre.SRE_Match at 0x38e4838>,
           <_sre.SRE_Match at 0x38e4870>]

In [208]: finditResult = re.finditer(r'\w', 'this is good day')

          for res in finditResult:
              print res.group()

t
h
i
s
i
s
g
o
o
d
d
a
y


In [211]: map(lambda x:x.group(), re.finditer(r'\w', 'this is good day'))

Out[211]: ['t', 'h', 'i', 's', 'i', 's', 'g', 'o', 'o', 'd', 'd', 'a', 'y']
```

```
In [212]: sentence = '''
          Today is 2nd week of this month.. 23 people came
          for this 16th  class.'''

In [213]: sentence

Out[213]: '\nToday is 2nd week of this month.. 23 people came \nfor this 16th  class.'

In [214]: re.findall('\d+', sentence)

Out[214]: ['2', '23', '16']

In [215]: finditResult = re.finditer('\d+', sentence)

In [216]: type(finditResult)

Out[216]: callable-iterator

In [217]: for mt in finditResult:
              print mt.span(), mt.group()

(10, 11) 2
(35, 37) 23
(60, 62) 16
```

1.11 Compilation Flags

```
IGNORECASE, I   - Des case-insensitive matches i.e, matches both upper and lowercase alphabets
DOTALL,     S   - Matches any character, including newlines.
MULTILINE,  M   - Multi-line matching, affecting ^ and $ re operators.
LOCALE,     L   - Does a locale-aware match.
VERBOSE,    X   - Enable verbose REs, which can be organized more cleanly and understandably.
UNICODE,    U   - Makes several escapes like \w, \b, \s and \d dependent on the unicode charac
```

re.I is shortform of re.IGNORECASE

```
In [222]: re.findall('python', 'Pythonpython, PYTHON pyTHoniC')

Out[222]: ['python']

In [223]: re.findall('python', 'Pythonpython, PYTHON pyTHoniC', re.I)

Out[223]: ['Python', 'python', 'PYTHON', 'pyTHon']

In [224]: re.findall('python','Pythonpython, PYTHON pyTHoniC', re.IGNORECASE)

Out[224]: ['Python', 'python', 'PYTHON', 'pyTHon']
```

```
In [241]: paragraph = '''
          This is good day.
          In this month, there are 31 days.
          this is 2nd week of July.
          thIS
          '''

In [242]: print paragraph


This is good day.
In this month, there are 31 days.
this is 2nd week of July.
thIS



In [243]: paragraph

Out[243]: '\nThis is good day.\nIn this month, there are 31 days.\nthis is 2nd week of July.\nt

In [244]: re.findall('this', paragraph)

Out[244]: ['this', 'this']

In [248]: re.findall('this', paragraph, re.M)

Out[248]: ['this', 'this']

In [249]: re.findall('this', paragraph, re.I)

Out[249]: ['This', 'this', 'this', 'thIS']

In [250]: re.findall('this', paragraph, re.I|re.M)

Out[250]: ['This', 'this', 'this', 'thIS']
```

1.12 re.ASCII() vs re.UNICODE() vs re.LOCALE()

re.ASCII is used to match all ASCII characters
re.UNICODE is used to match all unicode characters
re.LOCALE is used to match any local language (non-english) characters
re.UNICODE and re.LOCALE are used together

```
In [251]: re.findall('\w+', 'this is an example')

Out[251]: ['this', 'is', 'an', 'example']
```

Execute the above statement with a non-english language, if your system supports.

```
In [295]: re.findall(ur'\w+', ur''''''', re.UNICODE) # chinese
```

```
Out[295]: [u'\u4f60\u597d']

In [263]: re.findall(ur'\w+', ur'', re.UNICODE) # Telugu

Out[263]: [u'\u0c39\u0c32']
```

re.VERBOSE is used to create more readable re object. But, it works same as without placing this flag.

```
In [254]: reObj = re.compile(r'''
          &[#]              # Start of a numeric entity reference
          (
              0[0-7]+       # Octal form
           |  [0-9]+        # Decimal form
           |  x[0-9a-fA-F]+ # Hexadecimal form
          )
          ;                 # Trailing semicolon
          ''', re.VERBOSE)

In [255]: print reObj

<_sre.SRE_Pattern object at 0x02486860>


In [256]: reObj

Out[256]: re.compile(r'\n&[#]              # Start of a numeric entity reference\n(\n    0[0-7
          re.VERBOSE)

In [258]: # with out verbose flag, it wi look like
          reObj = re.compile('''
          &[#]              # Start of a numeric entity reference
          (
              0[0-7]+       # Octal form
           |  [0-9]+        # Decimal form
           |  x[0-9a-fA-F]+ # Hexadecimal form
          )
          ;                 # Trailing semicolon
          ''')

In [259]: print reObj

<_sre.SRE_Pattern object at 0x039C5780>


In [260]: reObj

Out[260]: re.compile(r'\n&[#]              # Start of a numeric entity reference\n(\n    0[0-7

In [261]: re.compile(r'[a-f|3-8]', re.DEBUG)  # to debug a pattern
```

```
in
  range (97, 102)
  literal 124
  range (51, 56)
```

```
Out[261]: re.compile(r'[a-f|3-8]', re.DEBUG)
```

\b word boundary - This is a zero-width assertion that matches only at the beginning or end of a word.

```
In [271]: p = re.compile(r'\bclass\b')

In [272]: p.search('declassified algorithm is classified in class')

Out[272]: <_sre.SRE_Match at 0x38dbaa0>

In [273]: p.search('declassified algorithm is classified in class').group()

Out[273]: 'class'

In [274]: p.search('some \bclass\b data').group()

Out[274]: 'class'

In [276]: p = re.compile('\bclass\b')

In [277]: p.search('declassified algorithm is classified in class').group()

        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-277-d9bcc9609a90> in <module>()
    ----> 1 p.search('declassified algorithm is classified in class').group()


        AttributeError: 'NoneType' object has no attribute 'group'


In [278]: p.search('some \bclass\b data').group()

Out[278]: '\x08class\x08'
```

\B works opposite to that of \b.

1.13 Result Modifiers

1.13.1 split()

- Split the string into alist, splitting it wherever the RE matches

```
In [279]: sentence = "It's theright right! right?"

In [280]: p = re.compile(r'\W+') #\W matches for absence of word

In [281]: p.split(sentence)

Out[281]: ['It', 's', 'theright', 'right', 'right', '']

In [283]: re.findall(r'\w+', sentence)
                  # In this case, it is same as split()

Out[283]: ['It', 's', 'theright', 'right', 'right']

In [284]: p.split(sentence, 3)
                 # Additionally, split() has option t specify  maxsplits

Out[284]: ['It', 's', 'theright', 'right! right?']

In [285]: p.split(sentence, 2)

Out[285]: ['It', 's', 'theright right! right?']
```

NOTE: length of result will be maxsplits + 1, when maxsplits !=0

```
In [287]: p.split(sentence,0) # default option

Out[287]: ['It', 's', 'theright', 'right', 'right', '']
```

1.13.2 sub()

- Results all substrings where the RE matches, and replaces them with a different string.

```
In [288]: p = re.compile('(blue|white|red)')

In [289]: p.sub('colour', 'blue Lorries and red Busses')

Out[289]: 'colour Lorries and colour Busses'

In [290]: p.sub('colour', 'blue Lorries and red Busses', count=1)

Out[290]: 'colour Lorries and red Busses'
```

1.13.3 subn()

- does the same job as sub; But, returns a tuple conatining the new strinng value, and the no. of replacements performed.

```
In [291]: p.subn('colour', 'blue lorries and red buses') # returns a tuple

Out[291]: ('colour lorries and colour buses', 2)

In [292]: p.subn('colour', 'no colous at all')

Out[292]: ('no colous at all', 0)
```

1.14 Summary of all the regular expressions

Non-special chars match themselves. Exceptions are special characters::

```
\       Escape special char or start a sequence.
.       Match any char except newline, see re.DOTALL
^       Match start of the string, see re.MULTILINE
$       Match end of the string, see re.MULTILINE
[]      Enclose a set of matchable chars
R|S     Match either regex R or regex S.
()      Create capture group, & indicate precedence
```

After '[', enclose a set, the only special chars are::

```
]   End the set, if not the 1st char
-   A range, eg. a-c matches a, b or c
^   Negate the set only if it is the 1st char
```

Quantifiers (append '?' for non-greedy)::

```
{m}     Exactly m repetitions
{m,n}   From m (default 0) to n (default infinity)
*       0 or more. Same as {,}
+       1 or more. Same as {1,}
?       0 or 1. Same as {,1}
```

Special sequences::

```
\A  Start of string
\b  Match empty string at word (\w+) boundary
\B  Match empty string not at word boundary
\d  Digit
\D  Non-digit
\s  Whitespace [ \t\n\r\f\v], see LOCALE,UNICODE
\S  Non-whitespace
\w  Alphanumeric: [0-9a-zA-Z_], see LOCALE
\W  Non-alphanumeric
```

```
\Z  End of string
\g<id>  Match prev named or numbered group,
        '<' & '>' are literal, e.g. \g<0>
        or \g<name> (not \g0 or \gname)
```

Special character escapes are much like those already escaped in Python string literals. Hence regex '\n' is same as regex '\\n'::

```
\a  ASCII Bell (BEL)
\f  ASCII Formfeed
\n  ASCII Linefeed
\r  ASCII Carriage return
\t  ASCII Tab
\v  ASCII Vertical tab
\\  A single backslash
\xHH   Two digit hexadecimal character goes here
\OOO   Three digit octal char (or just use an
       initial zero, e.g. \0, \09)
\DD    Decimal number 1 to 99, match
       previous numbered group
```

Extensions. Do not cause grouping, except 'P<name>'::

```
(?iLmsux)     Match empty string, sets re.X flags
(?:...)       Non-capturing version of regular parens
(?P<name>...) Create a named capturing group.
(?P=name)     Match whatever matched prev named group
(?#...)       A comment; ignored.
(?=...)       Lookahead assertion, match without consuming
(?!...)       Negative lookahead assertion
(?<=...)      Lookbehind assertion, match if preceded
(?<!...)      Negative lookbehind assertion
(?(id)y|n)    Match 'y' if group 'id' matched, else 'n'
```

Flags for re.compile(), etc. Combine with '|'::

```
re.I == re.IGNORECASE   Ignore case
re.L == re.LOCALE       Make \w, \b, and \s locale dependent
re.M == re.MULTILINE    Multiline
re.S == re.DOTALL       Dot matches all (including newline)
re.U == re.UNICODE      Make \w, \b, \d, and \s unicode dependent
re.X == re.VERBOSE      Verbose (unescaped whitespace in pattern
                        is ignored, and '#' marks comment lines)
```

Module level functions::

```
compile(pattern[, flags]) -> RegexObject
match(pattern, string[, flags]) -> MatchObject
search(pattern, string[, flags]) -> MatchObject
```

```
    findall(pattern, string[, flags]) -> list of strings
    finditer(pattern, string[, flags]) -> iter of MatchObjects
    split(pattern, string[, maxsplit, flags]) -> list of strings
    sub(pattern, repl, string[, count, flags]) -> string
    subn(pattern, repl, string[, count, flags]) -> (string, int)
    escape(string) -> string
    purge() # the re cache
```

RegexObjects (returned from `compile()`)::

```
.match(string[, pos, endpos]) -> MatchObject
.search(string[, pos, endpos]) -> MatchObject
.findall(string[, pos, endpos]) -> list of strings
.finditer(string[, pos, endpos]) -> iter of MatchObjects
.split(string[, maxsplit]) -> list of strings
.sub(repl, string[, count]) -> string
.subn(repl, string[, count]) -> (string, int)
.flags      # int, Passed to compile()
.groups     # int, Number of capturing groups
.groupindex # {}, Maps group names to ints
.pattern    # string, Passed to compile()
```

```
MatchObjects (returned from ``match()`` and ``search()``)::

.expand(template) -> string, Backslash & group expansion
.group([group1...]) -> string or tuple of strings, 1 per arg
.groups([default]) -> tuple of all groups, non-matching=default
.groupdict([default]) -> {}, Named groups, non-matching=default
.start([group]) -> int, Start/end of substring match by group
.end([group]) -> int, Group defaults to 0, the whole match
.span([group]) -> tuple (match.start(group), match.end(group))
.pos       int, Passed to search() or match()
.endpos    int, "
.lastindex int, Index of last matched capturing group
.lastgroup string, Name of last matched capturing group
.re        regex, As passed to search() or match()
.string    string, "
```

"'

References: 1. python regex cheatsheet

1.15 Popular Regular expression Generators:

1. http://regexr.com/
2. https://regex101.com/#python
3. http://www.regular-expressions.info/python.html

1.16 Popular Regular Expression Visualizers :

1. https://regexper.com

1.17 Popular RegEx creating Tools:

1. kodos (http://kodos.sourceforge.net/home.html)
    - Tool for computing and practicing regular expressions.
2. pythex(http://pythex.org/online)
    - regex generator created in python.