this notebook

Regular Expressions

Thursday, March 21, 2019 11:56 AM

Creating Regex Objects:

- Regular exp are available in re module. So need to import the same.
- .compile is a method which returns a regex object.
- .search is a method to search for the pattern passed.
- .group is a method which provides the match found. Below is an example of identifying regex way to find pattern matching in a string.

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> print('Phone number found: ' + mo.group())
Phone number found: 415-555-4242
```

- In the above example .compile of takes the pattern and returns a regex object.
- Now in the object search is performed by passing the string and result is stored in a regex variable.
- Now .group() method is applied to obtain the actual value.

Review of Regular Expression Matching :

While there are several steps to using regular expressions in Python, each
step is fairly simple.

1. Import the regex module with import re.
2. Create a Regex object with the re.compile() function. (Remember to use a raw string.)
3. Pass the string you want to search into the Regex object’s search() method. This returns a Match object.
4. Call the Match object’s group() method to return a string of the actual matched text.

Grouping with Parentheses :

- Used to group the output obtained from the regex exp.
    - group(0) / group () --> returns the entire result
    - group(1) --> returns the first group defined in the parenthesis. See below example.
    - group(2) --> returns the second group and so on.
    - groups() --> returns all the groups at once as two separate values. See example below.

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> mo.group(1)
'415'
>>> mo.group(2)
'555-4242'
>>> mo.group(0)
'415-555-4242'
>>> mo.group()
'415-555-4242'
>>> mo.groups()
('415', '555-4242')
>>> areaCode, mainNumber = mo.groups()
>>> print(areaCode)
415
>>> print(mainNumber)
555-4242
```

- If an expression has parenthesis then use **\(** and **\)** to match the pattern. An example is provided below.

```
>>> phoneNumRegex = re.compile(r'(\(\d\d\d\)) (\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My phone number is (415) 555-4242.')
>>> mo.group(1)
'(415)'
>>> mo.group(2)
'555-4242'
```

Matching Multiple Groups with the Pipe (OR) :

- If we want to match one or many exp use the | operator for the same.
- If the string passed contains both the expressions only the first matched exp will be returned by group() method.
- Can use findall() to return all the values of the searched expression.
- An example is provided for this scenario.

```
>>> heroRegex = re.compile (r'Batman|Tina Fey')
>>> mo1 = heroRegex.search('Batman and Tina Fey.')
>>> mo1.group()
'Batman'
>>> mo2 = heroRegex.search('Tina Fey and Batman.')
>>> mo2.group()
'Tina Fey'
```

- A part of the string can also be matched with as given below.

```
>>> batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
>>> mo = batRegex.search('Batmobile lost a wheel')
>>> mo.group()
'Batmobile'
```

```
>>> mo.group(1)
'mobile'
```

Optional Matching with the Question Mark (?) :

- Used when you want to match a pattern optionally. i.e. the exp may be an option in the given list.
- In the example provided below (wo) is optional and hence in the example both Batman as well as Batwoman are returned.

```
>>> batRegex = re.compile(r'Bat(wo)?man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'
>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
```

- In our phone number example sometimes it may be possible that area code is not specified which means that it is optional so the same can be written as given below.

```
>>> phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d\d')
>>> mo1 = phoneRegex.search('My number is 415-555-4242')
>>> mo1.group()
'415-555-4242'
>>> mo2 = phoneRegex.search('My number is 555-4242')
>>> mo2.group()
'555-4242'
```

Matching Zero or More with the Star (*):

- The above example only matches 0 or 1 of the group preceding the **?** mark. i.e. if Batwowowowwoman is entered the same will not be returned.
- To overcome the above situation we use * instead of ? Which returns 0 or more occurrence that precedes the *
- An example is given below.

```
>>> batRegex = re.compile(r'Bat(wo)*man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'
>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
>>> mo3 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

Matching One or More with the Plus (+) :

- While the above expressions doesn't mandate that the exp specified in the () should appear atleast one time, this one does.
- i.e. if the pattern match is performed with a plus it requires that at least once the pattern given the () should be present in the string.
- Below is an example of the same.

```
>>> batRegex = re.compile(r'Bat(wo)+man')
>>> mo1 = batRegex.search('The Adventures of Batwoman')
>>> mo1.group()
'Batwoman'
>>> mo2 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo2.group()
'Batwowowowoman'
>>> mo3 = batRegex.search('The Adventures of Batman')
>>> mo3 == None
True
```

- In the above example the last object i.e. mo3 returns None as **'wo'** is not present in the string.

Matching Specific Repetitions with Curly Brackets {} :

- If you want to match an exp specific number of times the same can be done with the help of {} .
- Enter the number of times the repetition occurs in {}.
- Following are the few ways in which pattern matching can be done.
  - (Ha){3} --> will match HaHaha.
  - (Ha){3,5} --> will match 'HaHaHa', 'HaHaHaHa', and 'HaHaHaHaHa'
  - (Ha) {3,} --> will match three or more instances of the (Ha) group
  - (Ha) {, 5} --> will match 0 to 5 instances of Ha in group.

```
>>> haRegex = re.compile(r'(Ha){3}')
>>> mo1 = haRegex.search('HaHaHa')
>>> mo1.group()
'HaHaHa'
>>> mo2 = haRegex.search('Ha')
>>> mo2 == None
True
```

- In general python searches for the longest string possible in a pattern match.
- This can be amended by adding a ? After the curly braces as given below. Here check the difference between the greedy and non-greedy versions.

```
>>> greedyHaRegex = re.compile(r'(Ha){3,5}')
>>> mo1 = greedyHaRegex.search('HaHaHaHaHa')
>>> mo1.group()
'HaHaHaHaHa'

>>> nongreedyHaRegex = re.compile(r'(Ha){3,5}?')
>>> mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
>>> mo2.group()
'HaHaHa'
```

The findall() Method :

- The search() of regex returns only the first matched text while findall() returns all the matched texts.

- When called on a regex with no groups, such as \d\d\d-\d\d\d-\d\d\d\d, the method findall() returns a list of string matches, such as ['415-555-9999', '212-555-0000'].

- When called on a regex that has groups, such as (\d\d\d)-(\d\d\d)-(\d\d\d\d), the method findall() returns a list of tuples of strings (one string for each group), such as [('415', '555', '1122'), ('212', '555', '0000')].

Making Your Own Character Classes :

- Our own character classes can be made as given below.

  vowelRegex = re.compile(r'[aeiouAEIOU]') --> will search for all the vowels
  alphanumericRegex = re.compile(r'[a-zA-Z0-9]') --> will search for alphabets and numbers
  consonantRegex = re.compile(r'[^aeiouAEIOU]') --> will search for all characters which are not vowels.

- **'^'** the caret sign can be used to indicate that the match must occur at the beginning of the search.
- **'$'** the dollar sign can be used to indicate that the match must occur at the end of the search.
  - beginsWithHello = re.compile(r'^Hello')
  - endsWithNumber = re.compile(r'\d$')
- **'.'** the dot character is the wildcard and will match any character except for a newline.

  ```
  >>> atRegex = re.compile(r'.at')
  >>> atRegex.findall('The cat in the hat sat on the flat mat.')
  ['cat', 'hat', 'sat', 'lat', 'mat']
  ```

- **'.*'** matches anything and everything after the given pattern.
- **Example 1**
  ```
  >>> nameRegex = re.compile(r'First Name: (.*) Last Name: (.*)')
  ```

```
>>> mo = nameRegex.search('First Name: Al Last Name: Sweigart')
>>> mo.group(1)
'Al'
>>> mo.group(2)
'Sweigart'
```

- **Example 2**

```
>>> nongreedyRegex = re.compile(r'<.*?>')
>>> mo = nongreedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man>'
>>> greedyRegex = re.compile(r'<.*>')
>>> mo = greedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man> for dinner.>'
```

- By passing re.DOTALL in the complie method for .* even newlines will be matched.

```
>>> noNewlineRegex = re.compile('.*')
>>> noNewlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.'
>>> newlineRegex = re.compile('.*', re.DOTALL)
>>> newlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

SUMMARY :

•The ? matches zero or one of the preceding group.
• The * matches zero or more of the preceding group.
• The + matches one or more of the preceding group.
• The {n} matches exactly n of the preceding group.
• The {n,} matches n or more of the preceding group.
• The {,m} matches 0 to m of the preceding group.
• The {n,m} matches at least n and at most m of the preceding group.
• {n,m}? or *? or +? performs a nongreedy match of the preceding group.
• ^spam means the string must begin with spam.
• spam$ means the string must end with spam.
• The . matches any character, except newline characters.
• \d, \w, and \s match a digit, word, or space character, respectively.
• \D, \W, and \S match anything except a digit, word, or space character, respectively.
• [abc] matches any character between the brackets (such as a, b, or c).
• [^abc] matches any character that isn’t between the brackets
• re.I in compile method is used to perform case insensitive matching.