PYTHON – REGULAR EXPRESSION

https://docs.python.org/3/howto/regex.html

RE

- A highly specialized programming language embedded inside Python and made available through the **re** module.
- Allows you to specify rules for just about any set of possible strings that you want to match
  - Virtually anything in ASCII!
- You can also use REs to modify a string or to split it apart in various ways.

START WITH AN EXAMPLE

```
>>> sentence = ' I Alphonso live and learn 999
 seeing nature go astern'

>>> import re

>>> my_match=re.compile('[a-z]+')

>>> list1 = my_match.findall(sentence)


>>> print(list1)

['lphonso', 'live', 'and', 'learn', 'seeing',
 'nature', 'go', 'astern']
```

META CHARACTERS

. ^ $ * + ? { } [ ] \ | ( )

METACHARACTERS []

- Square brackets: specify a character class.
- [acx] #means match ANY ONE of these.
- [a-dA-D]
- [a-z]
- [^5] #negation

EXAMPLE

- Which one it is going to match?
  - Regex: [cbe]at
    - cat,
    - sat
    - beat
    - Bat
    - eat

EXERCISE

```
>>> my_match=re.compile('put your RE here')
>>> list1 = my_match.findall("put your text here")
```

- Create a loop that prompts the user for a sentence. The program prints back the sentences which have two or more instances of words that start with either ‘c’ or ‘h’.

ANSWER

```
import re

m=re.compile('[ch][a-z]*')

while True:
      L=[]
      S=input('\n Enter a sentence  ')
      L=m.findall(S)
      if L:
            if len(L) > 1:
            print(S)
```

METACHARACTER (ESCAPE) \

- Can be followed by various characters to signal various special sequences.
- Also used to escape metacharacters so you can still match them in patterns;
  - i.e. \[or \\.

PRE-DEFINED CLASSES

- \d = class [0-9].
- \D Any non-digit, same as class [^0-9].
- \s Any whitespace, same as class [\t\n\r\f\v].
- \S Any non-whitespace chr, same as class [^ \t\n\r\f\v].
- \w Any alphanumeric chr, same as class [a-zA-Z0-9_].
- \W Any non-alphanumeric chr, same as class [^a-zA-Z0-9_].

METACHARACTER (ESCAPE) \

- But remember that metacharacters (like '.' and \) must be delimited if you are actually searching for the period or slash!

- To search for c:\readme.txt -> "c:\\readme\.txt"

EXERCISE

- write a program that prompts the user for a string and looks for all instances of ‘\section’

THE BACKSLASH PLAGUE

Characters	Stage
\section	Text string to be matched
\\section	Escaped backslash for re.compile()
"\\\\section"	Escaped backslashes for a string literal

Regular String	Raw string
"ab*"	r"ab*"
"\\\\section"	r"\\section"
"\\w+\\s+\\1"	r"\w+\s+\1"

METACHARACTER *

- Repeating things:
- “*” means 0 or more times
- “*” is said to be greedy!

EXAMPLE

- Consider the RE expression a[bcd]*b against the string “abcbd”:

Step	Matched	Explanation
1	a	The a in the RE matches.
2	abcbd	The engine matches [bcd]*, going as far as it can, which is to the end of the string.
3	*Failure*	The engine tries to match b, but the current position is at the end of the string, so it fails.
4	abcb	Back up, so that [bcd]* matches one less character.
5	*Failure*	Try b again, but the current position is at the last character, which is a 'd'.
6	abc	Back up again, so that [bcd]* is only matching bc.
6	abcb	Try b again. This time the character at the current position is 'b', so it succeeds.

METACHARACTERS + AND ?

- “+” matches one or more times.
  - requires at least one occurrence
  - Example: “ca+t”

- “?” matches either zero or one time
  - Example: “home-?brew”

EXERCISE

a) Write a program that prompts the user for a line of text then extracts all of the dot decimal address within that sentence.

b) Same as a) but now only retrieve addresses if the are in 192.168/16

Use: compile, findall, and an RE using metacharacters ‘\d’ and +

EXERCISE

```
import re

ma=re.compile('\d+\.\d+\.\d+\.\d+')
mb=re.compile('192\.168\.\d+\.\d+')

while True:
    l=[]
    x=input('\n Enter a line ')
    l=ma.findall(x)
    if l:
        for i in l:
            print(i)
```

.*

- Using ".*" can be dangerous.
- The quantifier is "greedy" and will match as much text as possible.
- To make a quantifier "non-greedy" (matching as few characters as possible), add a "?" after the "*".

- text='www.mysite.com then blah blah and finally ftp.example.com blah'

- Applied to the example above, "www\.my.*?\.com" would match just "www.mysite.com", not the longer string.

REPEATED QUALIFIER: {M,N}

- must be at least M repetitions, and at most N.

- Example: a/{1,3}b

METACHARACTER: |

- The “or” operator.
- If A and B are regular expressions, A|B will match any string that matches either A or B.
- | has very low precedence in order to make it work reasonably when you’re alternating multi-character strings. Crow|Servo will match either Crow or Servo only
- To match a literal '|', use \|, or enclose it inside a character class, as in [|].

METACHARACTER: ^

- Matches at the beginning of lines.
- Unless the MULTILINE flag has been set, this will only match at the beginning of the string.

```
print re.search('^From', 'From Here to Eternity')
      <_sre.SRE_Match object at 0x...>

print re.search('^From', 'Reciting From Memory')
      None
```

METACHARACTER: $

- Matches at the end of a line
- Defined as either the end of the string, or any location followed by a newline character.

```
print re.search('}$', '{block}')
      <_sre.SRE_Match object at 0x...>

print re.search('}$', '{block} ')
      None

print re.search('}$', '{block}\n')
      <_sre.SRE_Match object at 0x...>
```

Metacharacter: \b

Word boundary.

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

A word is defined as a sequence of alphanumeric characters, so the end of a word is indicated by whitespace or a non-alphanumeric character.

```
p = re.compile(r'\bclass\b')


print p.search('no class at all')
      <_sre.SRE_Match object at 0x...>


print p.search('the declassified
algorithm')
      None
```

COLLISION

- This is the worst collision between Python's string literals and regular expression sequences.
- In Python's string literals, \b is the backspace character, ASCII value 8.
- If you're not using raw strings, then Python will convert the \b to a backspace, and your RE won't match as you expect it to

```
p = re.compile('\bclass\b')


print p.search('no class at all')
      None
```

PARENTHESES "()"

- Used to group characters and expressions within larger, more complex regular expressions.
- Quantifiers that immediately follow the group apply to the whole group.

  - (abc){2,3}
  - abcabc
  - abcabcabc
  - abc
  - abccc

Exercise

```
R = '^\s*\w+'
```

```
R = '^\s*(\w+)\W+(\w+)\s*$'
```

Exercise

```
R = '(\w+)[.!?]?\s*$'

R = '^\d+$|^0x[\da-f]+$'

R = '\w\W'
```

Exercise

```
st = '/user/Jackie/temp/names.dat';
p = re.compile(r'^.*/(.*) ')
print (p.search(st).group(1))
```

Grouping revisited

- Frequently you need to obtain more information than just whether the RE matched or not.
- Regular expressions are often used to dissect strings by writing a RE divided into several subgroups which match different components of interest

Example

Review an RFC looking for headers with colons and capture the text group on either side of the colon

From: author@example.com

User-Agent: Thunderbird 1.5.0.9 (X11/20061227)

MIME-Version: 1.0

To: editor@example.com

Grouping

- Groups are marked by the round bracket metacharacters.

- () have much the same meaning as they do in mathematical expressions; they group together the expressions contained inside them
- You can repeat the contents of a group with a repeating qualifier, such as *, +, ?, or {m,n}.
- (ab)+
- (a|b)+
- (a|bb)+
- (a|ab)+

```
>>> p = re.compile('(ab)*')
>>> p.match('abababababab').span()
(0, 10)
```

Example

```
>>> text = 'one two three four:five and six: and
seven:eight'

>>> import re

>>> p = re.compile(r'((\w+):\w+)')

>>> p.findall(text)

[('four:five', 'four'), ('seven:eight', 'seven')]
```

Grouping

- Groups indicated with ( ) also capture the starting and ending index of the text that they match; this can be retrieved by passing an argument to `group()`, `start()`, `end()`, and `span()`.

- Groups are numbered starting with 0. Group 0 is always present; it's the whole RE, so MatchObject methods all have group 0 as their default argument.

Example

```
>>> p = re.compile('(a)b')
>>> m = p.match('ab')
>>> m.group()
'ab'
>>> m.group(0)
'ab'
>>> m.group(1)
‘a’
```

Groups

- Subgroups are numbered from left to right, from 1 upward.
- Groups can be nested; to determine the number, just count the opening parenthesis characters, going from left to right.

```
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

Groups

- `group()` can be passed multiple group numbers at a time, in which case it will return a tuple containing the corresponding values for those groups.

```
>>> m.group(2,1,2)
('b', 'abc', 'b')
```

- The `groups()` method returns a tuple containing the strings for all the subgroups, from 1 up to however many there are.

```
>>> m.groups()
('abc', 'b')
```

Exercise

```
>>> re2='(\d\d)(\d\d\d)(\d\d\d\d)'
>>> re2c=re.compile(re2)
>>> re2c.search('123456789876543244').group()
```

PERFORMING MATCHES

Method/Attribute	Purpose
match()	Determine if the RE matches at the beginning of the string.
search()	Scan through a string, looking for any location where this RE matches.
findall()	Find all substrings where the RE matches, and returns them as a list.
finditer()	Find all substrings where the RE matches, and returns them as an iterator.

TYPICAL CODE

```
p = re.compile( ... )
m = p.match( 'string' or string variable here)
if m:
    print 'Match found: ', m.group()
else:
    print 'No match'
```