Regular Expressions II

re.search(), re.compile() and the compile object

re.search() is the one-step method we've been using to test matching. Actually, regex matching is done in two steps: *compiling* and *searching*. **re.search()** conveniently puts the two together.

In some cases, a pattern should be compiled first before matching begins. This would be useful if the pattern is to be matched on a great number of strings, as in this weblog example:

```
import re
access_log = '/home1/d/dbb212/public_html/python/examples/access_log'
weblog = open(access_log)
patternobj = re.compile(r'edg205')
for line in weblog.readlines():
  if patternobj.search(line):
    print line,
weblog.close()
```

The **pattern object** is returned from **re.compile**, and can then be called with **search**. Here we're calling **search** repeatedly, so it is likely more efficient to compile once and then search with the compiled object.

Grouping for Alternates: Vertical Bar

We can group several characters together with parentheses. The parentheses do not affect the match, but they do designate a part of the matched string to be handled later. We do this to allow for alternate matches, for quantifying a portion of the pattern, or to extract text.

Inside a group, the vertical bar can indicate allowable matches. In this example, a string will match on any of these words, and because of the anchors will not allow any other characters:

```
r'^(y|yes|yeah|yep|yup|yu-huh)$'   # matches any of these words
```

A group may indicate alternating choices within a larger pattern:

```
r'(John|John D.)\s+Rockefeller'
                         # matches John Rockefeller or John D. Rockefeller
```

Grouping for Quantifying

Another reason to group would be to quantify a sequence of the pattern. For example, we could search for the "John Rockefeller or John D. Rockefeller" pattern by making the 'D.' optional:

```
r'John\s+(D\.\s+)?Rockefeller'
```

Grouping for Extraction: Memory Variables

We use the **group()** method of the **match** object to extract the text that matched the group:

```
string1 = "Find the nyu id, like dbb212, in this sentence"
matchobj = re.search(r'([a-z]{2,3}\d+)', string1)
id = matchobj.group(1)                          # id is 'dbb212'
```

Here's an example, using our log file. What if we wanted to capture the last two numbers (the status code and the number of bytes served), and place the values into structures?

```
log_lines = [
'66.108.19.165 - - [09/Jun/2003:19:56:33 -0400] "GET /~jjk265/cd.jpg HTTP/1.1" 200 175449',
'216.39.48.10 - - [09/Jun/2003:19:57:00 -0400] "GET /~rba203/about.html HTTP/1.1" 200 1566',
'216.39.48.10 - - [09/Jun/2003:19:57:16 -0400] "GET /~dd595/frame.htm HTTP/1.1" 400 1144'
]

import re
bytes_sum = 0
for line in log_lines:
  matchobj = re.search(r'(\d+) (\d+)$', line) # last two numbers in line
  status_code = matchobj.group(1)
  bytes = matchobj.group(2)
  bytes_sum += int(bytes)                    # sum the bytes
```

groups()

If you wish to grab all the matches into a tuple rather than call them by number, use **groups()**. You can then read variables from the tuple, or assign **groups()** to named variables, as in the last line below:

```
import re

name = "Richard M. Nixon"
matchobj = re.search(r'(\w+)\s+(\w)\.\s+(\w+)', name)
name_tuple = matchobj.groups()
print name_tuple    # ('Richard', 'M', 'Nixon')

(first, middle, last) = matchobj.groups()
print "%s, %s, %s" % ( first, middle, last )
```

flags: re.IGNORECASE

We can modify our matches with qualifiers called *flags*. The **re.IGNORECASE** flag will match any letters, whether upper or lowercase. In this example, extensions may be upper or lowercase - this file matcher doesn't care!

```
import re
dirlist = ('thisfile.jpg', 'thatfile.txt', 'otherfile.mpg', 'myfile.TXT')
for file in dirlist:
  if re.search(r'\.txt$', file, re.IGNORECASE):   #'.txt' or '.TXT'
    print file
```

The flag is passed as the third argument to **search**, and can also be passed to other **re** search methods.