Python 2.7 Regular Expressions

Special characters:

```
\       escapes special characters.
.       matches any character
^       matches start of the string (or line if MULTILINE)
$       matches end of the string (or line if MULTILINE)
[5b-d]  matches any chars '5', 'b', 'c' or 'd'
[^a-c6] matches any char except 'a', 'b', 'c' or '6'
R|S     matches either regex R or regex S.
()      Creates a capture group, and indicates precedence.
```

Within `[]`, no special chars do anything special, hence they don't need escaping, except for `']'` and `'-'`, which only need escaping if they are not the 1st char. e.g. `'[]]'` matches `']'`. `'^'` also has special meaning, it negates the group if it's the first character in the `[]`, and needs to be escaped to match it literally.

Quantifiers:

```
*       0 or more   (append ? for non-greedy)
+       1 or more    "
?       0 or 1       "
{m}     exactly 'm'
{m,n}   from m to n. 'm' defaults to 0, 'n' to infinity
{m,n}?  from m to n, as few as possible
```

Special sequences:

```
\A  Start of string
\b  Matches empty string at word boundary (between \w and \W)
\B  Matches empty string not at word boundary
\d  Digit
\D  Non-digit
\s  Whitespace: [ \t\n\r\f\v], more if LOCALE or UNICODE
\S  Non-whitespace
\w  Alphanumeric: [0-9a-zA-Z_], or is LOCALE dependant
\W  Non-alphanumeric
\Z  End of string

\g<id>  Match previous named or numbered group,
        e.g. \g<0> or \g<name>
```

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 Carraige return
\t  ASCII Tab
\v  ASCII Vertical tab
\\  A single backslash

\xHH    Two digit hex character
\OOO    Three digit octal char
        (or use a preceding zero, e.g. \0, \09)
\DD     Decimal number 1 to 99, matches previous
        numbered group
```

Extensions. These do not cause grouping, except for `(?P<name>...)`:

```
(?iLmsux)       Matches empty string, sets re.X flags
(?:...)         Non-capturing version of regular parentheses
(?P<name>...)   Creates a named capturing group.
(?P=name)       Matches whatever matched previously named group
(?#...)         A comment; ignored.
(?=...)         Lookahead assertion: Matches without consuming
(?!...)         Negative lookahead assertion
(?<=...)        Lookbehind assertion: Matches if preceded
(?<!...)        Negative lookbehind assertion
(?(id)yes|no)   Match 'yes' if group 'id' matched, else 'no'
```

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(pattner, 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 and group expansion
.group([group1...]) -> string or tuple of strings, 1 per arg
.groups([default]) -> (,) of all groups, non-matching=default
.groupdict([default]) -> {} of named groups, non-matching=default
.start([group]) -> int, start/end of substring matched by group
.end([group])      (group defaults to 0, the whole match)
.span([group]) -> tuple (match.start(group), match.end(group))
.pos # value passed to search() or match()
.endpos # "
.lastindex # int index of last matched capturing group
.lastgroup # string name of last matched capturing group
.re # regex passed to search() or match()
.string # string passed to search() or match()
```

Gleaned from the python 2.7 're' docs. http://docs.python.org/library/re.html

Version: v0.3.1

Contact: tartley@tartley.com