Class Regexp < Object

1.9

A Regexp holds a regular expression, used to match a pattern against strings. Regexps are created using the /.../ and %r... literals and using the Regexp.new constructor. Ruby 1.9 uses a different regular expression engine than previous versions.[4] See the reference on regular expressions starting 22 on page 332 for details.

Class constants

EXTENDED	Ignores spaces and newlines in regexp.
IGNORECASE	Matches are case insensitive.
MULTILINE	Newlines treated as any other character.

Class methods

compile

Regexp.compile(*pattern* ⟨ , *options* ⟨ , *lang* ⟩ ⟩) → *rxp*

Synonym for Regexp.new.

escape

Regexp.escape(*string*) → *escaped_string*

Escapes any characters that would have special meaning in a regular expression. For any string, Regexp.new(Regexp.escape(*str*))=~*str* will be true.

```
Regexp.escape('\\[]*?{}.')   # =>   \\\[\]\*\?\{\}\.
```

last_match

Regexp.last_match → *match*
Regexp.last_match(*int*) → *string*

The first form returns the MatchData object generated by the last successful pattern match. This is equivalent to reading the global variable $~. MatchData is described on page 585. The second form returns the n^{th} field in this MatchData object.

```
/c(.)t/ =~ 'cat'       # =>  0
Regexp.last_match      # =>  #<MatchData "cat" 1:"a">
Regexp.last_match(0)   # =>  "cat"
Regexp.last_match(1)   # =>  "a"
Regexp.last_match(2)   # =>  nil
```

new

Regexp.new(*string* ⟨ , *options* ⟨ , *lang* ⟩ ⟩) → *rxp*
Regexp.new(*regexp*) → *new_regexp*

Constructs a new regular expression from the *string* or the *regexp*. In the latter case, that regexp's options are propagated, and new options may not be specified. If *options* is a Fixnum, it should be one or more of Regexp::EXTENDED, Regexp::IGNORECASE, and Regexp::MULTILINE, *or*-ed together. Otherwise, if the *options* parameter is not nil, the regexp will be case insensitive. The *lang* can be set to "N" or "n" to force the regular expression

4. It is called *Oniguruma*.

to have ASCII-8BIT encoding;[5] otherwise, the encoding of the string determines the encoding of the regular expression.

```
# encoding: utf-8
r1 = Regexp.new('^[a-z]+:\\s+\w+')          # =>   /^[a-z]+:\s+\w+/
r2 = Regexp.new('cat', true)                # =>   /cat/i
r3 = Regexp.new('dog', Regexp::EXTENDED)    # =>   /dog/x
r4 = Regexp.new(r2)                         # =>   /cat/i
r5 = Regexp.new("δelta")                    # =>   /δelta/
r1.encoding                                 # =>   #<Encoding:US-ASCII>
r5.encoding                                 # =>   #<Encoding:UTF-8>
```

quote

Regexp.quote(*string*) → *escaped_string*

Synonym for Regexp.escape.

try_convert

Regexp.try_convert(*obj*) → *a_regexp* or nil

1.9

If *obj* is not already a regular expression, attempts to convert it to one by calling its to_regexp method. Returns nil if no conversion could be made.

```
Regexp.try_convert("cat")   # =>   nil
class String
  def to_regexp
    Regexp.new(self)
  end
end
Regexp.try_convert("cat")   # =>   /cat/
```

union

Regexp.union(⟨ pattern ⟩*) → *a_regexp*

1.9

Returns a regular expression that will match any of the given patterns. With no patterns, produces a regular expression that will never match. If a pattern is a string, it will be given the default regular expression options. If a pattern is a regular expression, its options will be honored in the final pattern. The patterns may also be passed in a single array.

```
Regexp.union("cat")           # =>   /cat/
Regexp.union("cat", "dog")    # =>   /cat|dog/
Regexp.union(%w{ cat dog })   # =>   /cat|dog/
Regexp.union("cat", /dog/i)   # =>   /cat|(?i-mx:dog)/
```

1.9

5. No other values are accepted as of Ruby 1.9.

Instance methods

== *rxp* == *other_regexp* → true or false

Equality—Two regexps are equal if their patterns are identical, they have the same character set code, and their casefold? values are the same.

```
/abc/  == /abc/x   # =>   false
/abc/  == /abc/i   # =>   false
/abc/u == /abc/n   # =>   false
```

=== *rxp* === *string* → true or false

Case Equality—Like Regexp#=~, but accepts nonstring arguments (returning false). Used in case statements.

```
a = "HELLO"
case a
when /^[a-z]*$/; print "Lower case\n"
when /^[A-Z]*$/; print "Upper case\n"
else             print "Mixed case\n"
end
```

produces:

```
Upper case
```

=~ *rxp* =~ *string* → *int* or nil

Match—Matches *rxp* against *string*, returning the offset of the start of the match or nil if the match failed. Sets $~ to the corresponding MatchData or nil.

```
/SIT/  =~ "insensitive"   # =>   nil
/SIT/i =~ "insensitive"   # =>   5
```

~ ~ *rxp* → *int* or nil

Match—Matches *rxp* against the contents of $_. Equivalent to *rxp* =~ $_. You should be ashamed if you use this....

```
$_ = "input data"
~ /at/   # =>   7
```

casefold? *rxp*.casefold? → true or false

Returns the value of the case-insensitive flag. Merely setting the i option inside *rxp* does not set this flag.

```
/cat/.casefold?        # =>   false
/cat/i.casefold?       # =>   true
/(?i:cat)/.casefold?   # =>   false
```

encoding

rxp.encoding → *an_encoding*

1.9

Returns the character encoding for the regexp.

```
/cat/.encoding    # =>   #<Encoding:US-ASCII>
/cat/s.encoding   # =>   #<Encoding:Windows-31J>
/cat/u.encoding   # =>   #<Encoding:UTF-8>
```

fixed_encoding?

rxp.fixed_encoding? → true or false

1.9

A regular expression containing only 7-bit characters can be matched against a string in any encoding. In this case, fixed_encoding? returns false. Otherwise, it returns true.

```
/cat/.fixed_encoding?    # =>   false
/cat/s.fixed_encoding?   # =>   true
/cat/u.fixed_encoding?   # =>   true
```

match

rxp.match(*string*, *offset*=0) → *match* or nil
rxp.match(*string*, *offset*=0) {| *match* | *block* } → *obj*

1.9

1.9

Returns a MatchData object (see page 585) describing the match or nil if there was no match. This is equivalent to retrieving the value of the special variable $~ following a normal match. The match process will start at *offset* into *string*. If a block is given and the match is successful, the block will be invoked with the MatchData object, and the value returned by the block will be the value returned by match.

```
md = /(.)(d)(.)/.match("abcdefabcdef")
md            # =>   #<MatchData "cde" 1:"c" 2:"d" 3:"e">
md[1]         # =>   "c"
md.begin(1)   # =>   2
md = /(.)(d)(.)/.match("abcdedcba", 4)
md            # =>   #<MatchData "edc" 1:"e" 2:"d" 3:"c">
md.begin(1)   # =>   4

result = /(...)...(...)/.match("catanddog") do |md|
  md[1] + "&" +  md[2]
end
result        # =>   "cat&dog"
```

named_captures

rxp.named_captures → *hash*

1.9

Returns a hash whose keys are the names of captures and whose values are each an array containing the number of the capture in *rxp*.

```
/(?<a>.).(?<b>.)/.named_captures         # =>   {"a"=>[1], "b"=>[2]}
/(?<a>.)(.)(?<b>.)/.named_captures       # =>   {"a"=>[1], "b"=>[2]}
/(?<a>.)(?<b>.)(?<a>.)/.named_captures   # =>   {"a"=>[1, 3], "b"=>[2]}
```

names

rxp.names → *array*

1.9

Returns an array containing the names of captures in *rxp*.

```
/(.)(.)(.)/.names                # =>   []
/(?<first>.).(?<last>.)/.names   # =>   ["first", "last"]
```

options

rxp.options → *int*

Returns the set of bits corresponding to the options used when creating this Regexp (see Regexp.new for details). Note that additional bits may be set in the returned options: these are used internally by the regular expression code. These extra bits are ignored if the options are passed to Regexp.new.

```
# Let's see what the values are...
Regexp::IGNORECASE                  # =>  1
Regexp::EXTENDED                    # =>  2
Regexp::MULTILINE                   # =>  4

/cat/.options                       # =>  0
/cat/ix.options                     # =>  3
Regexp.new('cat', true).options     # =>  1
Regexp.new('cat', 0, 'n').options   # =>  32

r = /cat/ix
Regexp.new(r.source, r.options)     # =>  /cat/ix
```

source

rxp.source → *string*

Returns the original string of the pattern.

```
/ab+c/ix.source   # =>  "ab+c"
```

to_s

rxp.to_s → *string*

Returns a string containing the regular expression and its options (using the (?xx:yyy) notation). This string can be fed back in to Regexp.new to a regular expression with the same semantics as the original. (However, Regexp#== may not return true when comparing the two, because the source of the regular expression itself may differ, as the example shows.) Regexp#inspect produces a generally more readable version of *rxp*.

```
r1 = /ab+c/ix         # =>  /ab+c/ix
s1 = r1.to_s          # =>  "(?ix-m:ab+c)"
r2 = Regexp.new(s1)   # =>  /(?ix-m:ab+c)/
r1 == r2              # =>  false
r1.source             # =>  "ab+c"
r2.source             # =>  "(?ix-m:ab+c)"
```