

Home

(http://yoururlhere.com/)

Core 2.2.0

(http://yoururlhere.com/core-2.2.0)

Std-lib 2.2.0

(http://yoururlhere.com/stdlib-2.2.0)

Downloads

(http://yoururlhere.com/downloads)

There was 1 Ruby vulnerability reports in the last 14 days. 1 undetermined. Most recent: CVE-2016-7954. See details. (https://web.nvd.nist.gov/view/vuln/search-results?q=cves=on)

Home (./index.html)

Classes (./index.html#classes)

Methods (./index.html#methods)

In Files

re.c

Parent

Object (Object.html)

Methods

::compile (method-c-compile)

::escape (method-c-escape)

::last_match (method-c-last_match)

::new (method-c-new)

::quote (method-c-quote)

::try_convert (method-c-try_convert)

::union (method-c-union)

=== (method-i-3D-3D)

=== (method-i-3D-3D)

#=~ (method-i-3D-7E)

#casefold? (method-i-casefold-3F)

#encoding (method-i-encoding)

#eq? (method-i-eq-3F)

#fixed_encoding? (method-i-fixed_encoding-3F)

#hash (method-i-hash)

#inspect (method-i-inspect)

#match (method-i-match)

#named_captures (method-i-named_captures)

#names (method-i-names)

#options (method-i-options)

#source (method-i-to_s)

#to_s (method-i-to_s)

~- (method-i-7E)

Files

grammar.en.rdoc (./lib/rdoc/grammar_en_rdoc.html)

test.ja.rdoc (./test/rdoc/test_ja_rdoc.html)

contributing.rdoc (./doc/contributing_rdoc.html)

contributors.rdoc (./doc/contributors_rdoc.html)

drace_probes.rdoc (./doc/dtrace_probes_rdoc.html)

globals.rdoc (./doc/globals_rdoc.html)

keywords.rdoc (./doc/keywords_rdoc.html)

maintainers.rdoc (./doc/maintainers_rdoc.html)

marshal.rdoc (./doc/marshal_rdoc.html)

regexp.rdoc (./doc/regexp_rdoc.html)

security.rdoc (./doc/security_rdoc.html)

standard_library.rdoc (./doc/standard_library_rdoc.html)

syntax.rdoc (./doc/syntax_rdoc.html)

assignment.rdoc (./doc/syntax/assignment_rdoc.html)

calling_methods.rdoc (./doc/syntax/calling_methods_rdoc.html)

control_expressions.rdoc (./doc/syntax/control_expressions_rdoc.html)

exceptions.rdoc (./doc/syntax/exceptions_rdoc.html)

literals.rdoc (./doc/syntax/literals_rdoc.html)

methods.rdoc (./doc/syntax/methods_rdoc.html)

miscellaneous.rdoc (./doc/syntax/miscellaneous_rdoc.html)

Regexp

A Regexp holds a regular expression, used to match a pattern against strings. Regexp's are created using the `/.../` and `%r{...}` literals, and by the `Regexp.new` constructor.

Regular expressions (*regexps*) are patterns which describe the contents of a string. They're used for testing whether a string contains a given pattern, or extracting the portions that match. They are created with the `/pat/` and `%r{pat}` literals or the `Regexp.new` constructor.

A regexp is usually delimited with forward slashes (`/`). For example:

```
/hay/ =~ 'haystack' #=> 0
/y/.match('haystack') #=> #<MatchData "y">
```

If a string contains the pattern it is said to *match*. A literal string matches itself.

Here 'haystack' does not contain the pattern 'needle', so it doesn't match:

```
/needle/.match('haystack') #=> nil
```

Here 'haystack' contains the pattern 'hay', so it matches:

```
/hay/.match('haystack') #=> #<MatchData "hay">
```

Specifically, `/st/` requires that the string contains the letter *s* followed by the letter *t*, so it matches *haystack*, also.

`==` and `#match` (Regexp.html#method-i-match) (#class-Regexp-label-3D-7E+and+Regexp-23match) ↑ (#top)

Pattern matching may be achieved by using `==` operator or `#match` (Regexp.html#method-i-match) method.

`==` operator (#class-Regexp-label-3D-7E+operator) ↑ (#top)

`==` is Ruby's basic pattern-matching operator. When one operand is a regular expression and the other is a string then the regular expression is used as a pattern to match against the string. (This operator is equivalently defined by `Regexp` (Regexp.html) and `String` (String.html) so the order of `String` (String.html) and `Regexp` (Regexp.html) do not matter. Other classes may have different implementations of `==`.) If a match is found, the operator returns index of first match in string, otherwise it returns `nil`.

```
/hay/ =~ 'haystack' #=> 0
'haystack' =~ /hay/ #=> 0
/a/ =~ 'haystack' #=> 1
/u/ =~ 'haystack' #=> nil
```

Using `==` operator with a `String` (String.html) and `Regexp` (Regexp.html) the `$~` global variable is set after a successful match. `$~` holds a `MatchData` (MatchData.html) object. `::last_match` (Regexp.html#method-c-last_match) is equivalent to `$~`.

`#match` (Regexp.html#method-i-match) method (#class-Regexp-label-Regexp-23match+method) ↑ (#top)

The `match` (Regexp.html#method-i-match) method returns a `MatchData` (MatchData.html) object:

```
/st/.match('haystack') #=> #<MatchData "st">
```

Metacharacters and Escapes (#class-Regexp-label-Metacharacters+and+Escapes) ↑ (#top)

The following are *metacharacters* `()[]{}.,*?`. They have a specific meaning when appearing in a pattern. To match them literally they must be backslash-escaped. To match a backslash literally backslash-escape that: `\\`.

```
/1 \+ 2 = 3\?/.match('Does 1 + 2 = 3?') #=> #<MatchData "1 + 2 = 3?">
```

Patterns behave like double-quoted strings so can contain the same backslash escapes.



```
modules_and_classes.rdoc (/doc/syntax
/modules_and_classes_rdoc.html)
precedence.rdoc (/doc/syntax/precedence_rdoc.html)
refinements.rdoc (/doc/syntax/refinements_rdoc.html)
README.ja.rdoc (/sample
/drb/README_ja_rdoc.html)
README.rdoc (/sample/drb/README_rdoc.html)
```

Class/Module Index

ArgumentError (/ArgumentError.html)
 Array (/Array.html)
 BasicObject (/BasicObject.html)



(http://www.cafepress.com/rubystuff)
 RubyStuff.com: The best Ruby T-shirts,
 mugs, bags, and mor
 (http://www.cafepress.com/rubystuff)

Encoding::Converter (/Encoding/Converter.html)
 Encoding::ConverterNotFoundError (/Encoding/Conve
 Encoding::InvalidByteSequenceError (/Encoding/Inval
 Encoding::UndefinedConversionError (/Encoding/Unde
 EncodingError (/EncodingError.html)
 Enumerable (/Enumerable.html)
 Enumerator (/Enumerator.html)
 Enumerator::Generator (/Enumerator/Generator.html)
 Enumerator::Lazy (/Enumerator/Lazy.html)
 Enumerator::Yielder (/Enumerator/Yielder.html)
 Ermo (/Ermno.html)
 Exception (/Exception.html)
 FalseClass (/FalseClass.html)
 Fiber (/Fiber.html)
 FiberError (/FiberError.html)
 File (/File.html)
 File::Constants (/File/Constants.html)
 File::Stat (/File/Stat.html)
 FileTest (/FileTest.html)
 Fixnum (/Fixnum.html)
 Float (/Float.html)
 FloatDomainError (/FloatDomainError.html)
 GC (/GC.html)
 GC::Profiler (/GC/Profiler.html)
 Hash (/Hash.html)
 IO (/IO.html)
 IO::EAGAINWaitReadable (/IO/EAGAINWaitReadable
 IO::EAGAINWaitWritable (/IO/EAGAINWaitWritable.h
 IO::EINPROGRESSWaitReadable (/IO/EINPROGRES
 IO::EINPROGRESSWaitWritable (/IO/EINPROGRES
 IO::EWOULDBLOCKWaitReadable (/IO/EWOULDB
 IO::EWOULDBLOCKWaitWritable (/IO/EWOULDB
 IO::WaitReadable (/IO/WaitReadable.html)
 IO::WaitWritable (/IO/WaitWritable.html)
 IOError (/IOError.html)
 IndexError (/IndexError.html)
 Integer (/Integer.html)
 Interrupt (/Interrupt.html)
 Kernel (/Kernel.html)
 KeyError (/KeyError.html)
 LoadError (/LoadError.html)
 LocalJumpError (/LocalJumpError.html)
 Marshal (/Marshal.html)
 MatchData (/MatchData.html)
 Math (/Math.html)
 Math::DomainError (/Math/DomainError.html)
 Method (/Method.html)
 Module (/Module.html)
 Mutex (/Mutex.html)
 NameError (/NameError.html)
 NilClass (/NilClass.html)
 NoMemoryError (/NoMemoryError.html)
 NoMethodError (/NoMethodError.html)
 NotImplementedError (/NotImplementedError.html)
 Numeric (/Numeric.html)
 Object (/Object.html)
 ObjectSpace (/ObjectSpace.html)
 ObjectSpace::WeakMap (/ObjectSpace/WeakMap.html)
 Proc (/Proc.html)
 Process (/Process.html)
 Process::GID (/Process/GID.html)
 Process::Status (/Process/Status.html)
 Process::Sys (/Process/Sys.html)
 Process::UID (/Process/UID.html)
 Process::Waiter (/Process/Waiter.html)
 Queue (/Queue.html)
 Random (/Random.html)

```
/\s\u{6771 4eac 90fd}/.match("Go to 東京都")

#=> #<MatchData " 東京都">
```

Arbitrary Ruby expressions can be embedded into patterns with the `#{...}` construct.

```
place = "東京都"

/#{place}/.match("Go to 東京都")

#=> #<MatchData "東京都">
```

Character Classes¶ (#class-Regexp-label-Character+Classes) ↑ (#top)

A *character class* is delimited with square brackets (`[,]`) and lists characters that may appear at that point in the match. `[ab]/` means *a* or *b*, as opposed to `/ab/` which means *a* followed by *b*.

```
/W[aeiou]rd/.match("Word") #=> #<MatchData "Word">
```

Within a character class the hyphen (`-`) is a metacharacter denoting an inclusive range of characters. `[abcd]` is equivalent to `[a-d]`. A range can be followed by another range, so `[abcdwxyz]` is equivalent to `[a-dw-z]`. The order in which ranges or individual characters appear inside a character class is irrelevant.

```
/[0-9a-f]/.match('9f') #=> #<MatchData "9">

/[9f]/.match('9f') #=> #<MatchData "9">
```

If the first character of a character class is a caret (`^`) the class is inverted: it matches any character *except* those named.

```
/[^a-eg-z]/.match('f') #=> #<MatchData "f">
```

A character class may contain another character class. By itself this isn't useful because `[a-z[0-9]]` describes the same set as `[a-z0-9]`. However, character classes also support the `&&` operator which performs set intersection on its arguments. The two can be combined as follows:

```
/[a-w&[~c-g]z]/ # {[a-w] AND ([~c-g] OR z)}
```

This is equivalent to:

```
/[abh~w]/
```

The following metacharacters also behave like character classes:

- `/./` - Any character except a newline.
- `/./m` - Any character (the `m` modifier enables multiline mode)
- `/\w/` - A word character (`[a-zA-Z0-9_]`)
- `/\W/` - A non-word character (`[^a-zA-Z0-9_]`). Please take a look at Bug #4044 (<https://bugs.ruby-lang.org/issues/4044>) if using `/\w/` with the `/i` modifier.
- `/\d/` - A digit character (`[0-9]`)
- `/\D/` - A non-digit character (`[^0-9]`)
- `/\h/` - A hexdigit character (`[0-9a-fA-F]`)
- `/\H/` - A non-hexdigit character (`[^0-9a-fA-F]`)
- `/\s/` - A whitespace character: `/[\t\r\n\f]/`
- `/\S/` - A non-whitespace character: `/[^ \t\r\n\f]/`

POSIX *bracket expressions* are also similar to character classes. They provide a portable alternative to the above, with the added benefit that they encompass non-ASCII characters. For instance, `/\d/` matches only the ASCII decimal digits (0-9); whereas `/[[[:digit:]]/` matches any character in the Unicode *Nd* category.

- `/[[[:alnum:]]/` - Alphabetic and numeric character
- `/[[[:alpha:]]/` - Alphabetic character
- `/[[[:blank:]]/` - Space or tab
- `/[[[:cntrl:]]/` - Control character
- `/[[[:digit:]]/` - Digit
- `/[[[:graph:]]/` - Non-blank character (excludes spaces, control characters, and similar)
- `/[[[:lower:]]/` - Lowercase alphabetical character
- `/[[[:print:]]/` - Like `[[:graph:]]`, but includes the space character
- `/[[[:punct:]]/` - Punctuation character
- `/[[[:space:]]/` - Whitespace character (`[[:blank:]]`, newline, carriage return, etc.)
- `/[[[:upper:]]/` - Uppercase alphabetical
- `/[[[:xdigit:]]/` - Digit allowed in a hexadecimal number (i.e., 0-9a-fA-F)

Ruby also supports the following non-POSIX character classes:

- `/[[[:word:]]/` - A character in one of the following Unicode general categories *Letter*, *Mark*, *Number*, *Connector_Punctuation*
- `/[[[:ascii:]]/` - A character in the ASCII character set

```
# U+06F2 is "EXTENDED ARABIC-INDIC DIGIT TWO"

/[[[:digit:]]/.match("\u06F2") #=> #<MatchData "\u{06F2}">

/[[[:upper:]]][[:lower:]]/.match("Hello") #=> #<MatchData "He">

/[[[:xdigit:]]][[:xdigit:]]/.match("A6") #=> #<MatchData "A6">
```

Repetition¶ (#class-Regexp-label-Repetition) ↑ (#top)

The constructs described so far match a single character. They can be followed by a repetition metacharacter to specify how many times they need to occur. Such metacharacters are called *quantifiers*.

- `*` - Zero or more times

[Range](#) ([./Range.html](#))
[RangeError](#) ([./RangeError.html](#))
[Rational](#) ([./Rational.html](#))
[Rational::compatible](#) ([./Rational/compatible.html](#))
[Regexp](#) ([./Regexp.html](#))
[RegexpError](#) ([./RegexpError.html](#))
[RubyVM](#) ([./RubyVM.html](#))
[RubyVM::Env](#) ([./RubyVM/Env.html](#))
[RubyVM::InstructionSequence](#) ([./RubyVM/InstructionSequence.html](#))
[RuntimeError](#) ([./RuntimeError.html](#))
[ScriptError](#) ([./ScriptError.html](#))
[SecurityError](#) ([./SecurityError.html](#))
[Signal](#) ([./Signal.html](#))
[SignalException](#) ([./SignalException.html](#))



<http://www.cafepress.com/rubystuff>
 RubyStuff.com: The best Ruby T-shirts,
 mugs, bags, and more
<http://www.cafepress.com/rubystuff>

[TracePoint](#) ([./TracePoint.html](#))
[TrueClass](#) ([./TrueClass.html](#))
[TypeError](#) ([./TypeError.html](#))
[UnboundMethod](#) ([./UnboundMethod.html](#))
[UncaughtThrowError](#) ([./UncaughtThrowError.html](#))
[ZeroDivisionError](#) ([./ZeroDivisionError.html](#))
[fatal](#) ([./fatal.html](#))
[unknown](#) ([./unknown.html](#))



- `+` - One or more times
- `?` - Zero or one times (optional)
- `{n}` - Exactly *n* times
- `{n,}` - *n* or more times
- `{,m}` - *m* or less times
- `{n,m}` - At least *n* and at most *m* times

At least one uppercase character ('H'), at least one lowercase character ('e'), two 'l' characters, then one 'o':

```
"Hello".match(/[[[:upper:]]+[[[:lower:]]+l(2)o/]) ==> #<MatchData "Hello">
```

Repetition is *greedy* by default: as many occurrences as possible are matched while still allowing the overall match to succeed. By contrast, *lazy* matching makes the minimal amount of matches necessary for overall success. A greedy metacharacter can be made lazy by following it with `?`.

Both patterns below match the string. The first uses a greedy quantifier so `'+'` matches `'<a>'`; the second uses a lazy quantifier so `'?+'` matches `'<a>'`:

```
/<.+>/.match("<a><b>") ==> #<MatchData "<a><b>">
```

```
/<.+?>/.match("<a><b>") ==> #<MatchData "<a>">
```

A quantifier followed by `+` matches *possessively*: once it has matched it does not backtrack. They behave like greedy quantifiers, but having matched they refuse to "give up" their match even if this jeopardises the overall match.

Capturing ¶ (#class-Regexp-label-Capturing) ↑ (#top)

Parentheses can be used for *capturing*. The text enclosed by the *n*th group of parentheses can be subsequently referred to with *n*. Within a pattern use the *backreference* `\n`; outside of the pattern use `MatchData[n]`.

'at' is captured by the first group of parentheses, then referred to later with `\1`:

```
/[csh](...) [csh]\1 in/.match("The cat sat in the hat")
```

```
#=> #<MatchData "cat sat in" 1:"at">
```

`#match` ([Regexp.html#method-i-match](#)) returns a `MatchData` ([MatchData.html](#)) object which makes the captured text available with its `[]` method:

```
/[csh](...) [csh]\1 in/.match("The cat sat in the hat")[1] ==> 'at'
```

Capture groups can be referred to by name when defined with the `(?<name>)` or `(?'name')` constructs.

```
/\$(?<dollars>\d+)\.(?<cents>\d+)/.match("$3.67")
```

```
=> #<MatchData "$3.67" dollars:"3" cents:"67">
```

```
/\$(?<dollars>\d+)\.(?<cents>\d+)/.match("$3.67")[:dollars] ==> "3"
```

Named groups can be backreferenced with `\k<name>`, where *name* is the group name.

```
/(?<vowel>[aeiou])\k<vowel>\k<vowel>/.match('ototomy')
```

```
#=> #<MatchData "ototo" vowel:"o">
```

Note: A regexp can't use named backreferences and numbered backreferences simultaneously.

When named capture groups are used with a literal regexp on the left-hand side of an expression and the `--` operator, the captured text is also assigned to local variables with corresponding names.

```
/\$(?<dollars>\d+)\.(?<cents>\d+)/ -- "$3.67" ==> 0
```

```
dollars ==> "3"
```

Grouping ¶ (#class-Regexp-label-Grouping) ↑ (#top)

Parentheses also *group* the terms they enclose, allowing them to be quantified as one *atomic* whole.

The pattern below matches a vowel followed by 2 word characters:

```
/[aeiou]\w{2}/.match("Caenorhabditis elegans") ==> #<MatchData "aen">
```

Whereas the following pattern matches a vowel followed by a word character, twice, i.e. `[aeiou]\w[aeiou]\w`: 'enor'.

```
/([aeiou]\w){2}/.match("Caenorhabditis elegans")
```

```
#=> #<MatchData "enor" 1:"or">
```

The `(?:...)` construct provides grouping without capturing. That is, it combines the terms it contains into an atomic whole without creating a backreference. This benefits performance at the slight expense of readability.

The first group of parentheses captures 'n' and the second 'ti'. The second group is referred to later with the backreference `\2`:

```
/I(n)ves(ti)ga\2ons/.match("Investigations")
```

```
#=> #<MatchData "Investigations" 1:"n" 2:"ti">
```

The first group of parentheses is now made non-capturing with `?:`, so it still matches 'n', but doesn't create the backreference. Thus, the backreference `\1` now refers to 'ti'.

```
/I(?:n)ves(ti)ga\lons/.match("Investigations")
#=> #<MatchData "Investigations" 1:"ti">
```

Atomic Grouping ¶ (#class-Regexp-label-Atomic+Grouping) ↑ (#top)

Grouping can be made *atomic* with `(?>pat)`. This causes the subexpression *pat* to be matched independently of the rest of the expression such that what it matches becomes fixed for the remainder of the match, unless the entire subexpression must be abandoned and subsequently revisited. In this way *pat* is treated as a non-divisible whole. Atomic grouping is typically used to optimise patterns so as to prevent the regular expression engine from backtracking needlessly.

The `"` in the pattern below matches the first character of the string, then `.*` matches `Quote`". This causes the overall match to fail, so the text matched by `.*` is backtracked by one position, which leaves the final character of the string available to match `"`

```
/" .*" /.match('"Quote"') #=> #<MatchData "\"Quote\"">
```

If `.*` is grouped atomically, it refuses to backtrack `Quote`", even though this means that the overall match fails

```
/" (?>.*)" /.match('"Quote"') #=> nil
```

Subexpression Calls ¶ (#class-Regexp-label-Subexpression+Calls) ↑ (#top)

The `\g<name>` syntax matches the previous subexpression named *name*, which can be a group name or number, again. This differs from backreferences in that it re-executes the group rather than simply trying to re-match the same text.

This pattern matches a `(` character and assigns it to the *paren* group, tries to call that the *paren* sub-expression again but fails, then matches a literal `)`:

```
/\A(?<paren>\(\g<paren>*\))\z/ -- '()' #=> 0
```

```
/\A(?<paren>\(\g<paren>*\))\z/ -- '()' #=> 0
# ^1
#   ^2
#     ^3
#       ^4
#         ^5
#           ^6
#             ^7
#               ^8
#                 ^9
#                   ^10
```

1. Matches at the beginning of the string, i.e. before the first character.
2. Enters a named capture group called *paren*
3. Matches a literal `(`, the first character in the string
4. Calls the *paren* group again, i.e. recurses back to the second step
5. Re-enters the *paren* group
6. Matches a literal `(`, the second character in the string
7. Try to call *paren* a third time, but fail because doing so would prevent an overall successful match
8. Match a literal `)`, the third character in the string. Marks the end of the second recursive call
9. Match a literal `)`, the fourth character in the string
10. Match the end of the string

Alternation ¶ (#class-Regexp-label-Alternation) ↑ (#top)

The vertical bar metacharacter `()` combines two expressions into a single one that matches either of the expressions. Each expression is an *alternative*.

```
/\w(and|or)\w/.match("Feliformia") #=> #<MatchData "form" 1:"or">
/\w(and|or)\w/.match("furandi") #=> #<MatchData "randi" 1:"and">
/\w(and|or)\w/.match("dissemlance") #=> nil
```

Character Properties ¶ (#class-Regexp-label-Character+Properties) ↑ (#top)

The `\p{}` construct matches characters with the named property, much like POSIX bracket classes.

- `\p{Alnum}` - Alphabetic and numeric character
- `\p{Alpha}` - Alphabetic character
- `\p{Blank}` - Space or tab
- `\p{Cntrl}` - Control character
- `\p{Digit}` - Digit
- `\p{Graph}` - Non-blank character (excludes spaces, control characters, and similar)
- `\p{Lower}` - Lowercase alphabetical character
- `\p{Print}` - Like `\p{Graph}`, but includes the space character
- `\p{Punct}` - Punctuation character
- `\p{Space}` - Whitespace character (`[:blank:]`, newline, carriage return, etc.)
- `\p{Upper}` - Uppercase alphabetical
- `\p{XDigit}` - Digit allowed in a hexadecimal number (i.e., 0-9a-fA-F)
- `\p{Word}` - A member of one of the following Unicode general category *Letter*, *Mark*, *Number*, *Connector*, *Punctuation*
- `\p{ASCII}` - A character in the ASCII character set
- `\p{Any}` - Any Unicode character (including unassigned characters)



- `/\p{Assigned}/` - An assigned character

A Unicode character's *General Category* value can also be matched with `\p{Ab}` where *Ab* is the category's abbreviation as described below:

- `/\p{L}/` - 'Letter'
- `/\p{Ll}/` - 'Letter: Lowercase'
- `/\p{Lm}/` - 'Letter: Mark'
- `/\p{Lo}/` - 'Letter: Other'
- `/\p{Lt}/` - 'Letter: Titlecase'
- `/\p{Lu}/` - 'Letter: Uppercase'
- `/\p{Lx}/` - 'Letter: Other'
- `/\p{M}/` - 'Mark'
- `/\p{Mn}/` - 'Mark: Nonspacing'
- `/\p{Mc}/` - 'Mark: Spacing Combining'
- `/\p{Me}/` - 'Mark: Enclosing'
- `/\p{N}/` - 'Number'
- `/\p{Nd}/` - 'Number: Decimal Digit'
- `/\p{Nl}/` - 'Number: Letter'
- `/\p{No}/` - 'Number: Other'
- `/\p{P}/` - 'Punctuation'
- `/\p{Pc}/` - 'Punctuation: Connector'
- `/\p{Pd}/` - 'Punctuation: Dash'
- `/\p{Ps}/` - 'Punctuation: Open'
- `/\p{Pe}/` - 'Punctuation: Close'
- `/\p{Pi}/` - 'Punctuation: Initial Quote'
- `/\p{Pf}/` - 'Punctuation: Final Quote'
- `/\p{Po}/` - 'Punctuation: Other'
- `/\p{S}/` - 'Symbol'
- `/\p{Sm}/` - 'Symbol: Math'
- `/\p{Sc}/` - 'Symbol: Currency'
- `/\p{Ss}/` - 'Symbol: Currency'
- `/\p{Sk}/` - 'Symbol: Modifier'
- `/\p{So}/` - 'Symbol: Other'
- `/\p{Z}/` - 'Separator'
- `/\p{Zs}/` - 'Separator: Space'
- `/\p{Zl}/` - 'Separator: Line'
- `/\p{Zp}/` - 'Separator: Paragraph'
- `/\p{C}/` - 'Other'
- `/\p{Cc}/` - 'Other: Control'
- `/\p{Cf}/` - 'Other: Format'
- `/\p{Cn}/` - 'Other: Not Assigned'
- `/\p{Co}/` - 'Other: Private Use'
- `/\p{Cs}/` - 'Other: Surrogate'



Lastly, `\p{}` matches a character's Unicode *script*. The following scripts are supported: *Arabic, Armenian, Balinese, Bengali, Bopomofo, Braille, Buginese, Buhid, Canadian_Aboriginal, Carian, Cham, Cherokee, Common, Coptic, Cuneiform, Cypriot, Cyrillic, Deseret, Devanagari, Ethiopic, Georgian, Glagolitic, Gothic, Greek, Gujarati, Gurmukhi, Han, Hangul, Hanunoo, Hebrew, Hiragana, Inherited, Kannada, Katakana, Kayah_Li, Kharoshthi, Khmer, Lao, Latin, Lepcha, Limbu, Linear_B, Lycian, Lydian, Malayalam, Mongolian, Myanmar, New_Tai_Lue, Nko, Ogham, Ol_Chiki, Old_Italic, Old_Persian, Oriya, Osmanya, Phags_Pa, Phoenician, Rejang, Runic, Saurashtra, Shavian, Sinhala, Sundanese, Syloti_Nagri, Syriac, Tagalog, Tagbanwa, Tai_Le, Tamil, Telugu, Thaana, Thai, Tibetan, Tifinagh, Ugaritic, Vai, and Yi.*

Unicode codepoint U+06E9 is named "ARABIC PLACE OF SAJDAH" and belongs to the Arabic script:

```
/\p{Arabic}/.match("\u06E9") #=> #<MatchData "\u06E9">
```

All character properties can be inverted by prefixing their name with a caret (^).

Letter 'A' is not in the Unicode Ll (Letter: Lowercase) category, so this match succeeds:

```
/\p{^Ll}/.match("A") #=> #<MatchData "A">
```

Anchors¶ (#class-Regexp-label-Anchors) ↑ (#top)

Anchors are metacharacter that match the zero-width positions between characters, *anchoring* the match to a specific position.

- `^` - Matches beginning of line
- `$` - Matches end of line
- `\A` - Matches beginning of string.
- `\Z` - Matches end of string. If string ends with a newline, it matches just before newline
- `\z` - Matches end of string
- `\G` - Matches point where last match finished
- `\b` - Matches word boundaries when outside brackets; backspace (0x08) when inside brackets
- `\B` - Matches non-word boundaries
- `(?=pat)` - *Positive lookahead* assertion: ensures that the following characters match *pat*, but doesn't include those characters in the matched text
- `(?!pat)` - *Negative lookahead* assertion: ensures that the following characters do not match *pat*, but doesn't include those characters in the matched text
- `(?<=pat)` - *Positive lookbehind* assertion: ensures that the preceding characters match *pat*, but doesn't include those characters in the matched text
- `(?<!pat)` - *Negative lookbehind* assertion: ensures that the preceding characters do not match *pat*, but doesn't include those characters in the matched text

If a pattern isn't anchored it can begin at any point in the string:

```
/real/.match("surrealist") #=> #<MatchData "real">
```

Anchoring the pattern to the beginning of the string forces the match to start there. 'real' doesn't occur at the

beginning of the string, so now the match fails:

```
/\Areal/.match("surrealist") #=> nil
```

The match below fails because although ‘Demand’ contains ‘and’, the pattern does not occur at a word boundary.

```
/\band/.match("Demand")
```

Whereas in the following example ‘and’ has been anchored to a non-word boundary so instead of matching the first ‘and’ it matches from the fourth letter of ‘demand’ instead:

```
/\Band.+/.match("Supply and demand curve") #=> #<MatchData "and curve">
```

The pattern below uses positive lookahead and positive lookbehind to match text appearing in tags without including the tags in the match:

```
/(?<=<b>)\w+(?=<\b>)/.match("Fortune favours the <b>bold</b>")
#=> #<MatchData "bold">
```

Options¶ (#class-Regexp-label-Options) ↑ (#top)

The end delimiter for a regexp can be followed by one or more single-letter options which control how the pattern can match.

- `/pat/i` - Ignore case
- `/pat/m` - Treat a newline as a character matched by `.`
- `/pat/x` - Ignore whitespace and comments in the pattern
- `/pat/o` - Perform `#{} interpolation` only once

`i`, `m`, and `x` can also be applied on the subexpression level with the *(?on-off)* construct, which enables options *on*, and disables options *off* for the expression enclosed by the parentheses.

```
/a(?i:b)c/.match('aBc') #=> #<MatchData "aBc">
/a(?i:b)c/.match('abc') #=> #<MatchData "abc">
```

Options may also be used with `Regexp.new`:

```
Regexp.new("abc", Regexp::IGNORECASE)      #=> /abc/i
Regexp.new("abc", Regexp::MULTILINE)       #=> /abc/m
Regexp.new("abc # Comment", Regexp::EXTENDED)  #=> /abc # Comment/x
Regexp.new("abc", Regexp::IGNORECASE | Regexp::MULTILINE)  #=> /abc/mi
```

Free-Spacing Mode and Comments¶ (#class-Regexp-label-Free-Spacing+Mode+and+Comments) ↑ (#top)

As mentioned above, the `x` option enables *free-spacing* mode. Literal white space inside the pattern is ignored, and the octothorpe (`#`) character introduces a comment until the end of the line. This allows the components of the pattern to be organized in a potentially more readable fashion.

A contrived pattern to match a number with optional decimal places:

```
float_pat = /\A
    [[:digit:]]+ # 1 or more digits before the decimal point
    (\.        # Decimal point
    [[:digit:]]+ # 1 or more digits after the decimal point
    )? # The decimal point and following digits are optional
\Z/x
float_pat.match('3.14') #=> #<MatchData "3.14" 1:".14">
```

There are a number of strategies for matching whitespace:

- Use a pattern such as `\s` or `\p{Space}`.
- Use escaped whitespace such as `\`, i.e. a space preceded by a backslash.
- Use a character class such as `[\]`.

Comments can be included in a non-`x` pattern with the *(?#comment)* construct, where *comment* is arbitrary text ignored by the regexp engine.

Encoding (Encoding.html)¶ (#class-Regexp-label-Encoding) ↑ (#top)

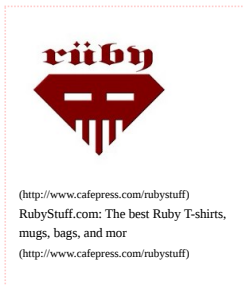
Regular expressions are assumed to use the source encoding. This can be overridden with one of the following modifiers.

- `/pat/u` - UTF-8
- `/pat/e` - EUC-JP
- `/pat/s` - Windows-31J
- `/pat/n` - ASCII-8BIT

A regexp can be matched against a string when they either share an encoding, or the regexp’s encoding is *US-ASCII* and the string’s encoding is ASCII-compatible.

If a match between incompatible encodings is attempted an `Encoding::CompatibilityError` exception is raised.

The `Regexp#fixed_encoding?` predicate indicates whether the regexp has a *fixed* encoding, that is one incompatible with ASCII. A regexp’s encoding can be explicitly fixed by supplying `Regexp::FIXEDENCODING` as



the second argument of Regexp.new:

```
r = Regexp.new("a".force_encoding("iso-8859-1"), Regexp::FIXEDENCODING)
r =~ "a\u3042"

#=> Encoding::CompatibilityError: incompatible encoding regexp match
      (ISO-8859-1 regexp with UTF-8 string)
```

Special global variables¶ (#class-Regexp-label-Special+global+variables) ↑ (#top)

Pattern matching sets some global variables :

- `$~` is equivalent to `::last_match (Regexp.html#method-c-last_match)`;
- `$&` contains the complete matched text;
- `$'` contains string before match;
- `$'` contains string after match;
- `$1`, `$2` and so on contain text matching first, second, etc capture group;
- `$+` contains last capture group.

Example:

```
m = /\s(\w{2}).*(c)/.match('haystack') #=> #<MatchData "stac" 1:"ta" 2:"c">

$~                                     #=> #<MatchData "stac" 1:"ta" 2:"c">
Regexp.last_match                     #=> #<MatchData "stac" 1:"ta" 2:"c">

$&      #=> "stac"
        # same as m[0]

$'       #=> "hay"
        # same as m.pre_match

$'       #=> "k"
        # same as m.post_match

$1       #=> "ta"
        # same as m[1]

$2       #=> "c"
        # same as m[2]

$3       #=> nil
        # no third group in pattern

$+       #=> "c"
        # same as m[-1]
```

These global variables are thread-local and method-local variables.

Performance¶ (#class-Regexp-label-Performance) ↑ (#top)

Certain pathological combinations of constructs can lead to abysmally bad performance.

Consider a string of 25 *as*, a *d*, 4 *as*, and a *c*.

```
s = 'a' * 25 + 'd' + 'a' * 4 + 'c'
#=> "aaaaaaaaaaaaaaaaaaaaaaaaadaaac"
```

The following patterns match instantly as you would expect:

```
/(b|a)/ =~ s #=> 0
/(b|a+)/ =~ s #=> 0
/(b|a+)* / =~ s #=> 0
```

However, the following pattern takes appreciably longer:

```
/(b|a+)*c/ =~ s #=> 26
```

This happens because an atom in the regexp is quantified by both an immediate `+` and an enclosing `*` with nothing to differentiate which is in control of any particular character. The nondeterminism that results produces super-linear performance. (Consult *Mastering Regular Expressions* (3rd ed.), pp 222, by Jeffery Friedl, for an in-depth analysis). This particular case can be fixed by use of atomic grouping, which prevents the unnecessary backtracking:

```
(start = Time.now) %$ /(b|a+)*c/ =~ s %$ (Time.now - start)
#=> 24.702736882

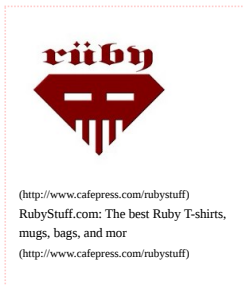
(start = Time.now) %$ /(b|a+)*c/ =~ s %$ (Time.now - start)
#=> 0.000166571
```

A similar case is typified by the following example, which takes approximately 60 seconds to execute for me:

Match a string of 29 *as* against a pattern of 29 optional *as* followed by 29 mandatory *as*:

```
Regexp.new('a?' * 29 + 'a' * 29) =~ 'a' * 29
```

The 29 optional *as* match the string, but this prevents the 29 mandatory *as* that follow from matching. Ruby must



then backtrack repeatedly so as to satisfy as many of the optional matches as it can while still matching the mandatory 29. It is plain to us that none of the optional matches can succeed, but this fact unfortunately eludes Ruby.

The best way to improve performance is to significantly reduce the amount of backtracking needed. For this case, instead of individually matching 29 optional *as*, a range of optional *as* can be matched all at once with *a{0,29}*:

```
Regexp.new('a{0,29}' + 'a' * 29) =~ 'a' * 29
```

Constants

EXTENDED

see #options (Regexp.html#method-i-options) and ::new (Regexp.html#method-c-new)

FIXEDENCODING

see #options (Regexp.html#method-i-options) and ::new (Regexp.html#method-c-new)

IGNORECASE

see #options (Regexp.html#method-i-options) and ::new (Regexp.html#method-c-new)

MULTILINE

see #options (Regexp.html#method-i-options) and ::new (Regexp.html#method-c-new)

NOENCODING

see #options (Regexp.html#method-i-options) and ::new (Regexp.html#method-c-new)

Public Class Methods

compile(*args)

Synonym for Regexp.new

escape(str) → string

Escapes any characters that would have special meaning in a regular expression. Returns a new escaped string, or self if no characters are escaped. For any string, Regexp.new(Regexp.escape(str))=str will be true.

```
Regexp.escape('\*?{,}') #=> "\\*\\?\\{\\}\\,"
```

last_match → matchdata

last_match(n) → str

The first form returns the MatchData (MatchData.html) object generated by the last successful pattern match. Equivalent to reading the special global variable \$~ (see Special global variables in Regexp (Regexp.html) for details).

The second form returns the *n*th field in this MatchData (MatchData.html) object. *n* can be a string or symbol to reference a named capture.

Note that the ::last_match (Regexp.html#method-c-last_match) is local to the thread and method scope of the method that did the pattern match.

```
/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

/(?<lhs>\w+)\s*=\s*(?<rhs>\w+)/ =~ "var = val"
Regexp.last_match     #=> #<MatchData "var = val" lhs:"var" rhs:"val">
Regexp.last_match(:lhs) #=> "var"
Regexp.last_match(:rhs) #=> "val"
```

new(string, [options [, kcode]]) → regexp

new(regexp) → regexp

compile(string, [options [, kcode]]) → regexp

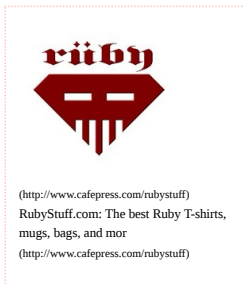
compile(regexp) → regexp

Constructs a new regular expression from pattern, which can be either a String (String.html) or a Regexp (Regexp.html) (in which case that regexp's options are propagated), and new options may not be specified (a change as of Ruby 1.8).

If options is a Fixnum (Fixnum.html), it should be one or more of the constants Regexp::EXTENDED, Regexp::IGNORECASE, and Regexp::MULTILINE, or-ed together. Otherwise, if options is not nil or false, the regexp will be case insensitive.

When the kcode parameter is 'n' or 'N' sets the regexp no encoding. It means that the regexp is for binary strings.

```
r1 = Regexp.new('a-z+:\s+\w+') #=> /a-z+:\s+\w+/
r2 = Regexp.new('cat', true)   #=> /cat/i
r3 = Regexp.new(r2)            #=> /cat/i
```

```
r4 = Regexp.new('dog', Regexp::EXTENDED | Regexp::IGNORECASE) #=> /dog/i
```

quote(str) → string

Escapes any characters that would have special meaning in a regular expression. Returns a new escaped string, or self if no characters are escaped. For any string, `Regexp.new(Regexp.escape(str))=str` will be true.

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

try_convert(obj) → re or nil

Try to convert *obj* into a Regexp (Regexp.html), using `to_regexp` method. Returns converted regexp or nil if *obj* cannot be converted for any reason.

```
Regexp.try_convert(/re/)      #=> /re/
Regexp.try_convert("re")     #=> nil
```

```
o = Object.new
Regexp.try_convert(o)         #=> nil
def o.to_regexp() /foo/ end
Regexp.try_convert(o)         #=> /foo/
```

union(pat1, pat2, ...) → new_regexp

union(pats_ary) → new_regexp

Return a Regexp object that is the union of the given *patterns*, i.e., will match any of its parts. The *patterns* can be Regexp (Regexp.html) objects, in which case their options will be preserved, or Strings. If no patterns are given, returns `/(?!)/`. The behavior is unspecified if any given *pattern* contains capture.

```
Regexp.union                #=> /(?!)/
Regexp.union("penzance")    #=> /penzance/
Regexp.union("a+b*c")       #=> /a+b*c/
Regexp.union("skiing", "sledding") #=> /skiing|sledding/
Regexp.union(["skiing", "sledding"]) #=> /skiing|sledding/
Regexp.union(/dogs/, /cats/) #=> /(?-mix:dogs)|(?i-mx:cats)/
```

Note: the arguments for `::union` (Regexp.html#method-c-union) will try to be converted into a regular expression literal via `to_regexp`.

Public Instance Methods

rxp == other_rxp → 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/ == /abc/u    #=> false
/abc/u == /abc/n   #=> false
```

rxp === str → true or false

Case Equality—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
#=> "Upper case"
```

Following a regular expression literal with the `===` (Regexp.html#method-i-3D-3D-3D) operator allows you to compare against a `String` (String.html).

```
/^[a-z]*/ === "HELLO" #=> false
/^[A-Z]*/ === "HELLO" #=> true
```

rxp =~ str → integer or nil

Match—Matches *rxp* against *str*.

```
/at/ =~ "input data" #=> 7
/ax/ =~ "input data" #=> nil
```

If `==` is used with a regexp literal with named captures, captured strings (or nil) is assigned to local variables named by the capture names.



```
/(?<lhs>\w+)\s*=(?<rhs>\w+)/ =~ " x = y "
p lhs    #=> "x"
p rhs    #=> "y"
```

If it is not matched, nil is assigned for the variables.

```
/(?<lhs>\w+)\s*=(?<rhs>\w+)/ =~ " x = "
p lhs    #=> nil
p rhs    #=> nil
```

This assignment is implemented in the Ruby parser. The parser detects 'regexp-literal =~ expression' for the assignment. The regexp must be a literal without interpolation and placed at left hand side.

The assignment does not occur if the regexp is not a literal.

```
re = /(?<lhs>\w+)\s*=(?<rhs>\w+)/
re =~ " x = y "
p lhs    # undefined local variable
p rhs    # undefined local variable
```

A regexp interpolation, # {}, also disables the assignment.

```
rhs_pat = /(?<rhs>\w+)/
/(?<lhs>\w+)\s*=#{rhs_pat}/ =~ "x = y"
p lhs    # undefined local variable
```

The assignment does not occur if the regexp is placed at the right hand side.

```
" x = y " =~ /(?<lhs>\w+)\s*=(?<rhs>\w+)/
p lhs, rhs # undefined local variable
```

casefold? → true or false

Returns the value of the case-insensitive flag.

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

encoding → encoding

Returns the Encoding (Encoding.html) object that represents the encoding of obj.

eql?(other_rxp) → 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/ == /abc/u    #=> false
/abc/u == /abc/n    #=> false
```

**fixed_encoding? → true or false**

Returns false if `rxp` is applicable to a string with any ASCII compatible encoding. Returns true otherwise.

```

r = /a/

r.fixed_encoding?           #=> false

r =~ "\u{6666} a"          #=> 2

r =~ "\xa1\xa2 a".force_encoding("euc-jp") #=> 2

r =~ "abc".force_encoding("euc-jp")        #=> 0


r = /a/u

r.fixed_encoding?           #=> true

r.encoding                  #=> #<Encoding:UTF-8>

r =~ "\u{6666} a"          #=> 2

r =~ "\xa1\xa2".force_encoding("euc-jp")   #=> ArgumentError

r =~ "abc".force_encoding("euc-jp")        #=> 0


r = /\u{6666}/

r.fixed_encoding?           #=> true

r.encoding                  #=> #<Encoding:UTF-8>

r =~ "\u{6666} a"          #=> 0

r =~ "\xa1\xa2".force_encoding("euc-jp")   #=> ArgumentError

r =~ "abc".force_encoding("euc-jp")        #=> nil

```

hash → fixnum

Produce a hash based on the text and options of this regular expression.

See also `Object#hash` ([Object.html#method-i-hash](#)).

inspect → string

Produce a nicely formatted string-version of `rxp`. Perhaps surprisingly, `#inspect` actually produces the more natural version of the string than `#to_s`.

```

/ab+c/ix.inspect           #=> "/ab+c/ix"

```

match(str) → matchdata or nil**match(str,pos) → matchdata or nil**

Returns a `MatchData` object 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. If the second parameter is present, it specifies the position in the string to begin the search.

```

/(.)(.)(.)/.match("abc")[2] #=> "b"

/(.)(.)(.)/.match("abc", 1)[2] #=> "c"

```

If a block is given, invoke the block with `MatchData` ([MatchData.html](#)) if match succeed, so that you can write

```

pat.match(str) {|m| ...}

```

instead of

```

if m = pat.match(str)
  ...
end

```

The return value is a value from block execution in this case.

named_captures → hash

Returns a hash representing information about named captures of `rxp`.

A key of the hash is a name of the named captures. A value of the hash is an array which is list of indexes of corresponding named captures.

```

/(?<foo>.)(<bar>.)/.named_captures

#=> {"foo"=>[1], "bar"=>[2]}


/(?<foo>.)(<foo>.)/.named_captures

#=> {"foo"=>[1, 2]}

```

If there are no named captures, an empty hash is returned.

```

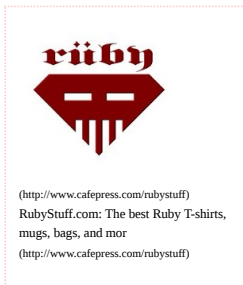
/(.)(.)/.named_captures

#=> {}

```

names → [name1, name2, ...]

Returns a list of names of captures as an array of strings.



```
/(?<foo>.)?(?<bar>.)?(?<baz>.)?/.names
#=> ["foo", "bar", "baz"]

/(?<foo>.)?(?<foo>.)?/.names
#=> ["foo"]

/(.)(.)/.names
#=> []
```

options → fixnum

Returns the set of bits corresponding to the options used when creating this Regexp ([Regexp.html](#)) (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`).

```
Regexp::IGNORECASE      #=> 1
Regexp::EXTENDED        #=> 2
Regexp::MULTILINE       #=> 4

/cat/.options           #=> 0
/cat/ix.options         #=> 3
Regexp.new('cat', true).options #=> 1
/\\xa1\\xa2\\xe.options  #=> 16

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

source → str

Returns the original string of the pattern.

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

Note that escape sequences are retained as is.

```
/\\x20+/.source #=> "\\x20\\x+"
```

to_s → str

Returns a string containing the regular expression and its options (using the `{?opts:source}` 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, as 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)"
```

~ rxp → integer or nil

Match—Matches `rxp` against the contents of `$_`. Equivalent to `rxp =~ $_`.

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

Additional notes

To learn more about Ruby regular expressions see:

The Bastards Book of Ruby - Regular Expressions (<http://ruby.bastardsbook.com/chapters/regexes/>)

Exploring Ruby's Regular Expression Algorithm (<http://patshaughnessy.net/2012/4/3/exploring-rubys-regular-expression-algorithm>)

Commenting is here to help enhance the documentation. For example, code samples, or clarification of the documentation.

If you have questions about Ruby or the documentation, please post to one of the **Ruby mailing lists** (<http://www.ruby-lang.org/en/community/mailling-lists/>). You will get better, faster, help that way.

If you wish to post a correction of the docs, please do so, but also [file bug report \(http://bugs.ruby-lang.org\)](http://bugs.ruby-lang.org) so that it can be corrected for the next release. Thank you.

If you want to help improve the Ruby documentation, please visit [Documenting-ruby.org \(http://documenting-ruby.org\)](http://documenting-ruby.org).

0 Comments

Ruby-doc.org

 Login








(<http://www.cafepress.com/rubystuff>)
RubyStuff.com: The best Ruby T-shirts,
mugs, bags, and mor
(<http://www.cafepress.com/rubystuff>)

re

Sort by Newest

cussion...

Be the first to comment.

 Add Disqus to your site  Add Disqus to your site  Add Disqus to your site  Add Disqus to your site  Add Disqus to your site  Add Disqus to your site  Privacy