Class String < Object

A String object holds and manipulates a sequence of bytes, typically representing characters. String objects may be created using String.new or as literals (see page 328).

Because of aliasing issues, users of strings should be aware of the methods that modify the contents of a String object. Typically, methods with names ending in ! modify their receiver, while those without a ! return a new String. However, exceptions exist, such as String#[]=.

In this description, I try to differentiate between the bytes in a string and the characters in a string. Internally, a string is a sequence of 8-bit bytes. These are represented externally as small Fixnums. At the same time, these byte sequences can be interpreted as a sequence of characters. This interpretation is controlled by the encoding of the string. In some encodings (such as US-ASCII and ISO-8859), each byte corresponds to a single character. In other encodings (such as UTF-8), a varying number of bytes comprise each character.

1.9 As of Ruby 1.9, String no longer mixes in Enumerable.

Mixes in

Comparable:

```
<, <=, ==, >=, >, between?
```

Class methods

new

String.new(*val*="") → *str*

Returns a new string object containing a copy of *val* (which should be a String or implement to_str). Note that the new string object is created only when one of the strings is modified.

```
str1 = "wibble"
str2 = String.new(str1)
str1.object_id   # =>   336070
str2.object_id   # =>   335970
str1[1] = "o"
str1             # =>   "wobble"
str2             # =>   "wibble"
```

try_convert

String.try_convert(*obj*) → *a_string* or nil

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

```
String.try_convert("cat")   # =>   "cat"
String.try_convert(0xbee)   # =>   nil
```

Instance methods

% *str % arg → string*

Format—Uses *str* as a format specification and returns the result of applying it to *arg*. If the format specification contains more than one substitution, then *arg* must be an Array containing the values to be substituted. See Kernel.sprintf on page 577 for details of the format string.

```
puts "%05d" % 123
puts "%-5s: %08x" % [ "ID", self.object_id ]
puts "%-5<name>s: %08<value>x" % { name: "ID", value: self.object_id }
```

produces:

```
00123
ID   : 000653ba
ID   : 000653ba
```

***** *str * int → string*

Copies—Returns a new String containing *int* copies of the receiver.

```
"Ho! " * 3   # =>   "Ho! Ho! Ho! "
```

+ *str + string → string*

Concatenation—Returns a new String containing *string* concatenated to *str*. If both strings contain non-7-bit characters, their encodings must be compatible.

```
"Hello from " + "RubyLand"   # =>   "Hello from RubyLand"
```

<< *str << fixnum → str*
str << obj → str

1.9 Append—Concatenates the given object to *str*. If the object is a Fixnum, it is considered to be a codepoint in the encoding of *str* and converted to the appropriate character before being appended.

```
a = "hello world"
a.force_encoding("utf-8")
a << 33             # =>   "hello world!"
a << " Says the "   # =>   "hello world! Says the "
a << 8706           # =>   "hello world! Says the ∂"
a << "og"           # =>   "hello world! Says the ∂og"
```

<=> *str <=> other_string → −1, 0, +1*

Comparison—Returns -1 if *str* is less than, 0 if *str* is equal to, and $+1$ if *str* is greater than *other_string*. If the strings are of different lengths and the strings are equal when compared up to the shortest length, then the longer string is considered greater than the shorter one. In

1.9 older versions of Ruby, setting $= allowed case-insensitive comparisons; you must now use String#casecmp.

<=> is the basis for the methods <, <=, >, >=, and between?, included from module Comparable. The method String#== does not use Comparable#==.

```
"abcdef" <=> "abcde"     # =>  1
"abcdef" <=> "abcdef"    # =>  0
"abcdef" <=> "abcdefg"   # =>  -1
"abcdef" <=> "ABCDEF"    # =>  1
```

== *str* == *obj* → true or false

Equality—If *obj* is a String, returns true if *str* has the same encoding, length, and content as *obj*; returns false otherwise. If *obj* is not a String but responds to to_str, returns *obj* == *str*; otherwise, returns false.

```
"abcdef" == "abcde"    # =>  false
"abcdef" == "abcdef"   # =>  true
```

=~ *str* =~ *regexp* → *int* or nil

Match—Equivalent to *regexp* =~ *str*. Prior versions of Ruby permitted an arbitrary operand to =~; this is now deprecated. Returns the position the match starts or returns nil if there is no match or if *regexp* is not a regular expression.[6]

```
"cat o' 9 tails" =~ /\d/   # =>  7
"cat o' 9 tails" =~ 9      # =>  nil

"cat o' 9 tails" =~ "\d"
```

produces:

```
prog.rb:1:in `=~': type mismatch: String given (TypeError)
from /tmp/prog.rb:1:in `<main>'
```

[ ]

str[*int*] → *string* or nil
str[*int*, *int*] → *string* or nil
str[*range*] → *string* or nil
str[*regexp*] → *string* or nil
str[*regexp*, *int*] → *string* or nil
str[*string*] → *string* or nil

1.9

Element Reference—If passed a single *int*, returns the character at that position. (Prior to Ruby 1.9, an integer character code was returned.) If passed two *ints*, returns a substring starting at the offset given by the first, and a length given by the second. If given a range, a substring containing characters at offsets given by the range is returned. In all three cases, if an offset is negative, it is counted from the end of *str*. Returns nil if the initial offset falls outside the string and the length is not given, the length is negative, or the beginning of the range is greater than the end.

If *regexp* is supplied, the matching portion of *str* is returned. If a numeric parameter follows the regular expression, that component of the MatchData is returned instead. If a String is given, that string is returned if it occurs in *str*. In both cases, nil is returned if there is no match.

6. Except for a strange corner case. If *regexp* is a string or can be coerced into a string, a TypeError exception is raised.

```
a = "hello there"
a[1]                   # =>  "e"
a[1,3]                 # =>  "ell"
a[1..3]                # =>  "ell"
a[1...3]               # =>  "el"
a[-3,2]                # =>  "er"
a[-4..-2]              # =>  "her"
a[-2..-4]              # =>  ""
a[/[aeiou](.)\1/]      # =>  "ell"
a[/[aeiou](.)\1/, 0]   # =>  "ell"
a[/[aeiou](.)\1/, 1]   # =>  "l"
a[/[aeiou](.)\1/, 2]   # =>  nil
a[/(..)e/]             # =>  "the"
a[/(..)e/, 1]          # =>  "th"
a["lo"]                # =>  "lo"
a["bye"]               # =>  nil
```

[ ]=

str[*int*] = *string*
str[*int*, *int*] = *string*
str[*range*] = *string*
str[*regexp*] = *string*
str[*regexp*, *int*] = *string*
str[*string*] = *string*

Element Assignment—Replaces some or all of the content of *str*. The portion of the string affected is determined using the same criteria as String#[]. If the replacement string is not the same length as the text it is replacing, the string will be adjusted accordingly. If the regular expression or string is used as the index doesn't match a position in the string, IndexError is raised. If the regular expression form is used, the optional second *int* allows you to specify which portion of the match to replace (effectively using the MatchData indexing rules). The forms that take a Fixnum will raise an IndexError if the value is out of range; the Range form will raise a RangeError, and the Regexp and String forms will silently ignore the assignment.

```
a = "hello"
a[2] = "u"                      (a → "heulo")
a[2, 4] = "xyz"                 (a → "hexyz")
a[-4, 2] = "xyz"                (a → "hxyzlo")
a[2..4] = "xyz"                 (a → "hexyz")
a[-4..-2] = "xyz"               (a → "hxyzo")
a[/[aeiou](.)\1(.)/] = "xyz"    (a → "hxyz")
a[/[aeiou](.)\1(.)/, 1] = "xyz" (a → "hexyzlo")
a[/[aeiou](.)\1(.)/, 2] = "xyz" (a → "hellxyz")
a["l"] = "xyz"                  (a → "hexyzlo")
a["ll"] = "xyz"                 (a → "hexyzo")
a[2, 0] = "xyz"                 (a → "hexyzllo")
```

ascii_only?

str.ascii_only? → true or false

1.9

Returns true if the string contains no characters with a character code greater than 127 (that is, it contains only 7-bit ASCII characters).

```
# encoding: utf-8
"dog".ascii_only?           # =>  true
"δog".ascii_only?           # =>  false
"\x00 to \x7f".ascii_only?  # =>  true
```

bytes

str.bytes → *enum*
str.bytes {| *byte* | *block* } → *str*

1.9

Returns an enumerator for the bytes (integers in the range 0 to 255) in *str*. With a block, passes each byte to the block and returns the original string. See also String#chars and String#codepoints.

```
# encoding: utf-8
"dog".bytes.to_a                      # =>  [100, 111, 103]
"δog".bytes.to_a                      # =>  [226, 136, 130, 111, 103]
result = []
"δog".bytes.each {|b| result << b }   # =>  "δog"
result                                # =>  [226, 136, 130, 111, 103]
```

bytesize

str.bytesize → *int*

1.9

Returns the number of bytes (not characters) in *str*. See also String#length.

```
# encoding: utf-8
"dog".length    # =>  3
"dog".bytesize  # =>  3
"δog".length    # =>  3
"δog".bytesize  # =>  5
```

capitalize

str.capitalize → *string*

Returns a copy of *str* with the first character converted to uppercase and the remainder to lowercase.

```
"hello world".capitalize  # =>  "Hello world"
"HELLO WORLD".capitalize  # =>  "Hello world"
"123ABC".capitalize       # =>  "123abc"
```

capitalize!

str.capitalize! → *str* or nil

Modifies *str* by converting the first character to uppercase and the remainder to lowercase. Returns nil if no changes are made.

```
a = "hello world"
a.capitalize!  # =>  "Hello world"
a              # =>  "Hello world"
a.capitalize!  # =>  nil
```

casecmp

str.casecmp(*string*) → −1, 0, +1

Case-insensitive version of String#<=>.

```
"abcdef".casecmp("abcde")     # =>  1
"abcdef".casecmp("abcdef")    # =>  0
"aBcDeF".casecmp("abcdef")    # =>  0
"abcdef".casecmp("abcdefg")   # =>  -1
"abcdef".casecmp("ABCDEF")    # =>  0
```

center

str.center(*int*, *pad*=" ") → *string*

If *int* is greater than the length of *str*, returns a new String of length *int* with *str* centered between the given padding (defaults to spaces); otherwise, returns *str*.

```
"hello".center(4)           # =>  "hello"
"hello".center(20)          # =>  "       hello        "
"hello".center(4,  "_-^-")  # =>  "hello"
"hello".center(20, "_-^-")  # =>  "_-^-_-^hello_-^-_-^-"
"hello".center(20, "-")     # =>  "-------hello--------"
```

chars

str.chars → *enum*

str.chars {| *char* | *block* } → *str*

1.9

Returns an enumerator for the characters (single character strings) in *str*. With a block, passes each character to the block and returns the original string. See also String#bytes.

```
# encoding: utf-8
"dog".chars.to_a                      # =>  ["d", "o", "g"]
"δog".chars.to_a                      # =>  ["δ", "o", "g"]
result = []
"δog".chars.each {|b| result << b }   # =>  "δog"
result                                # =>  ["δ", "o", "g"]
```

chr

str.chr → *string*

1.9

Returns the first character of *str*.

```
# encoding: utf-8
"dog".chr   # =>  "d"
"δog".chr   # =>  "δ"
```

clear

str.clear → *str*

1.9

Removes the content (but not the associated encoding) of *str*.

```
# encoding: utf-8
str = "δog"
str.clear      # =>  ""
str.length     # =>  0
str.encoding   # =>  #<Encoding:UTF-8>
```

chomp

str.chomp(*rs*=$/) → *string*

Returns a new String with the given record separator removed from the end of *str* (if

present). If $/ has not been changed from the default Ruby record separator, then chomp also removes carriage return characters (that is it will remove \n, \r, and \r\n).

```
"hello".chomp            # =>   "hello"
"hello\n".chomp          # =>   "hello"
"hello\r\n".chomp        # =>   "hello"
"hello\n\r".chomp        # =>   "hello\n"
"hello\r".chomp          # =>   "hello"
"hello \n there".chomp   # =>   "hello \n there"
"hello".chomp("llo")     # =>   "he"
```

chomp! *str*.chomp!(*rs*=$/) → *str* or nil

Modifies *str* in place as described for String#chomp, returning *str*or returning nil if no modifications were made.

chop *str*.chop → *string*

Returns a new String with the last character removed. If the string ends with \r\n, both characters are removed. Applying chop to an empty string returns an empty string. String#chomp is often a safer alternative, because it leaves the string unchanged if it doesn't end in a record separator.

```
"string\r\n".chop   # =>   "string"
"string\n\r".chop   # =>   "string\n"
"string\n".chop     # =>   "string"
"string".chop       # =>   "strin"
"x".chop.chop       # =>   ""
```

chop! *str*.chop! → *str* or nil

Processes *str* as for String#chop, returning *str* or returning nil if *str* is the empty string. See also String#chomp!.

codepoints *str*.codepoints → *enum*
str.codepoints {| *integer* | *block* } → *str*

1.9

Returns an enumerator for the codepoints (integers representation of the characters) in *str*. With a block, passes each integer to the block and returns the original string. See also String#bytes and String#chars.

```
# encoding: utf-8
"dog".codepoints.to_a                      # =>   [100, 111, 103]
"δog".codepoints.to_a                      # =>   [8706, 111, 103]
result = []
"δog".codepoints.each {|b| result << b }   # =>   "δog"
result                                     # =>   [8706, 111, 103]
```

concat *str*.concat(*int*) → *str*
str.concat(*obj*) → *str*

Synonym for String#<<.

count

str.count(⟨ *string* ⟩+) → *int*

Each *string* parameter defines a set of characters to count. The intersection of these sets defines the characters to count in *str*. Any parameter that starts with a caret (^) is negated. The sequence c_1–c_2 means all characters between c_1 and c_2.

```
a = "hello world"
a.count "lo"            # =>  5
a.count "lo", "o"       # =>  2
a.count "hello", "^l"   # =>  4
a.count "ej-m"          # =>  4
```

crypt

str.crypt(*settings*) → *string*

Applies a one-way cryptographic hash to *str* by invoking the standard library function crypt. The argument is to some extent system dependent. On traditional Unix boxes, it is often a two-character *salt* string. On more modern boxes, it may also control things such as DES encryption parameters. See the man page for crypt(3) for details.

```
# standard salt
"secret".crypt("sh")          # =>  "shRK3aVg8FsI2"
# On OSX: DES, 2 interactions, 24-bit salt
"secret".crypt("_...0abcd")   # =>  "_...0abcdROn65JNDj12"
```

delete

str.delete(⟨ *string* ⟩+) → *new_string*

Returns a copy of *str* with all characters in the intersection of its arguments deleted. Uses the same rules for building the set of characters as String#count.

```
"hello".delete("l","lo")        # =>  "heo"
"hello".delete("lo")            # =>  "he"
"hello".delete("aeiou", "^e")   # =>  "hell"
"hello".delete("ej-m")          # =>  "ho"
```

delete!

str.delete!(⟨ *string* ⟩+) → *str* or nil

Performs a delete operation in place, returning *str* or returning nil if *str* was not modified.

```
a = "hello"
a.delete!("l","lo")   # =>  "heo"
a                     # =>  "heo"
a.delete!("l")        # =>  nil
```

downcase

str.downcase → *string*

Returns a copy of *str* with all uppercase letters replaced with their lowercase counterparts. The operation is locale insensitive—only characters *A* to *Z* are affected. Multibyte characters are skipped.

```
"hEllO".downcase   # =>  "hello"
```

downcase! *str*.downcase! → *str* or nil

Replaces uppercase letters in *str* with their lowercase counterparts. Returns nil if no changes were made.

dump *str*.dump → *string*

Produces a version of *str* with all nonprinting characters replaced by \nnn notation and all special characters escaped.

each_byte *str*.each_byte → *enum*
str.each_byte {| *byte* | *block* } → *str*

1.9 Synonym for String#bytes. The each_byte form is falling out of favor.

each_char *str*.each_char → *enum*
str.each_char {| *char* | *block* } → *str*

1.9 Synonym for String#chars. The each_char form is falling out of favor.

each_codepoint *str*.each_codepoint → *enum*
str.each_codepoint {| *integer* | *block* } → *str*

1.9 Synonym for String#codepoints.

each_line *str*.each_line(*sep*=$/) → *enum*
str.each_line(*sep*=$/) {| *substr* | *block* } → *str*

1.9 Synonym for String#lines. The each_line form is falling out of favor.

empty? *str*.empty? → true or false

Returns true if *str* has a length of zero.

```
"hello".empty?   # =>   false
"".empty?        # =>   true
```

encode *str*.encode → *a_string*
str.encode(*to_encoding* ⟨ , options ⟩) → *a_string*
str.encode(*to_encoding*, *from_encoding*, ⟨ , options ⟩) → *a_string*

1.9 Transcodes *str*, returning a new string encoded as *to_encoding*. If no encoding is given, transcodes using default_internal encoding. The source encoding is either the current encoding of the string or *from_encoding*. May raise a RuntimeError if characters in the original string cannot be represented in the target encoding. *options* defines the behavior for invalid transcodings and other boundary conditions. It can be a hash or an or-ing of integer values. I recommend the hash form—see Table 27.15 on page 680 for details. Encodings can be passed as Encoding objects or as names.

```
# encoding: utf-8
ole_in_utf = "olé"
ole_in_utf.encoding    # =>   #<Encoding:UTF-8>
ole_in_utf.dump        # =>   "ol\u{e9}"

ole_in_8859 = ole_in_utf.encode("iso-8859-1")
ole_in_8859.encoding   # =>   #<Encoding:ISO-8859-1>
ole_in_8859.dump       # =>   "ol\xE9"
```

Using a default internal encoding of ISO-8859-1 and a source file encoding of UTF-8:

```
#!/usr/local/rubybook/bin/ruby -E:ISO-8859-1
# encoding: utf-8
utf_string = "olé"
utf_string.encoding   # =>   #<Encoding:UTF-8>
iso_string = utf_string.encode
iso_string.encoding   # =>   #<Encoding:ISO-8859-1>
```

Attempt to transcode a string with characters not available in the destination encoding:

```
# encoding: utf-8
utf = "δog"
utf.encode("iso-8859-1")
```

produces:

```
prog.rb:3:in `encode': "δ" from UTF-8 to ISO-8859-1
      (Encoding::UndefinedConversionError)
from /tmp/prog.rb:3:in `<main>'
```

You can replace the character in error with something else:

```
# encoding: utf-8
utf = "δog"
utf.encode("iso-8859-1", undef: :replace)                  # =>   "?og"
utf.encode("iso-8859-1", undef: :replace, replace: "X" )   # =>   "Xog"
```

encode!

str.encode! → *str*
str.encode!(*to_encoding* ⟨ , options ⟩) → *str*
str.encode!(*to_encoding*, *from_encoding*, ⟨ , options ⟩) → *str*

1.9

Transcodes *str* in place.

encoding

str.encoding → *an_encoding*

1.9

Returns the encoding of *str*.

```
# encoding: utf-8
"cat".encoding   # =>   #<Encoding:UTF-8>
"δog".encoding   # =>   #<Encoding:UTF-8>
```

end_with?

str.end_with?(⟨ suffix ⟩+) → true or false

1.9

Returns true if *str* ends with any of the given suffices.

```
"Apache".end_with?("ache")                       # =>   true
"ruby code".end_with?("python", "perl", "code")  # =>   true
```

Table 27.15. Options to Encode and Encode!

Option	Meaning
:replace => *string*	Specifies the string to use if :invalid or :undef options are present. If not specified, uFFFD is used for Unicode encodings and ? for others.
:invalid => :replace	Replaces invalid characters in the source string with the replacement string. If :invalid is not specified or nil, raises an exception.
:undef => :replace	Replaces characters that are not available in the destination encoding with the replacement string. If :undef not specified or nil, raises an exception.
:universal_newline => true	Converts crlf and cr line endings to lf.
:crlf_newline => true	Converts lf to crlf.
:cr_newline => true	Converts lf to cr.
:xml => :text | :attr	After encoding, escape characters that would otherwise have special meaning in XML PCDATA or attributes. In all cases, converts & to &, < to <, > to >, and undefined characters to a hexadecimal entity (&#xhh;). For :attr, also converts " to ".

eql?

str.eql?(*obj*) → true or false

Returns true if *obj* is a String with identical contents to *str*.

```
"cat".eql?("cat")   # =>   true
```

force_encoding

str.force_encoding(*encoding*) → *str*

1.9

Sets the encoding associated with *str* to *encoding*. Note that this does not change the underlying bytes in *str*—it simply tells Ruby how to interpret those bytes as characters.

```
# encoding: utf-8
δog_in_bytes = [226, 136, 130, 111, 103]  # utf-8 byte sequence
str = δog_in_bytes.pack("C*")
str.encoding   # =>   #<Encoding:ASCII-8BIT>
str.length     # =>   5
str.force_encoding("utf-8")
str.encoding   # =>   #<Encoding:UTF-8>
str.length     # =>   3
str            # =>   "δog"
```

getbyte

str.getbyte(*offset*) → *int* or nil

1.9

Returns the byte at *offset* (starting from the end of the string if the offset is negative). Returns nil if the offset lies outside the string.

```
# encoding: utf-8
str = "δog"
str.bytes.to_a    # =>  [226, 136, 130, 111, 103]
str.getbyte(0)    # =>  226
str.getbyte(1)    # =>  136
str.getbyte(-1)   # =>  103
str.getbyte(99)   # =>  nil
```

gsub

str.gsub(*pattern*, *replacement*) → *string*
str.gsub(*pattern*) {| *match* | *block* } → *string* *str*.gsub(*pattern*) → *enum*

1.9

Returns a copy of *str* with *all* occurrences of *pattern* replaced with either *replacement* or the value of the block. The *pattern* will typically be a Regexp; if it is a String, then no regular expression metacharacters will be interpreted (that is /\d/ will match a digit, but '\d' will match a backslash followed by a *d*).

If a string is used as the replacement, special variables from the match (such as $& and $1) cannot be substituted into it, because substitution into the string occurs before the pattern match starts. However, the sequences \1, \2, and so on, may be used to interpolate successive numbered groups in the match, and \k<*name*> will substitute the corresponding named captures. These sequences are shown in Table 27.16 on the following page.

In the block form, the current match is passed in as a parameter, and variables such as $1, $2, $`, $&, and $' will be set appropriately. The value returned by the block will be substituted for the match on each call.

The result inherits any tainting in the original string or any supplied replacement string.

```
"hello".gsub(/[aeiou]/, '*')                  # =>  "h*ll*"
"hello".gsub(/([aeiou])/, '<\1>')             # =>  "h<e>ll<o>"
"hello".gsub(/./) {|s| s[0].to_s + ' '}       # =>  "h e l l o "
"hello".gsub(/(?<double>l)/, '-\k<double>-')  # =>  "he-l--l-o"
```

1.9

If no block or replacement string is given, an enumerator is returned.

```
"hello".gsub(/../).to_a   # =>  ["he", "ll"]
```

1.9

If a hash is given as the replacement, successive matched groups are looked up as keys, and the corresponding values are substituted into the string.

```
repl = Hash.new("?")
repl["a"] = "*"
repl["t"] = "T"
"cat".gsub(/(.)/, repl)   # =>  "?*T"
```

gsub!

str.gsub!(*pattern*, *replacement*) → *str* or nil
str.gsub!(*pattern*) {| *match* | *block* } → *str* or nil

Performs the substitutions of String#gsub in place, returning *str*, or returning nil if no substitutions were performed.

Table 27.16. Backslash Sequences in Substitution Strings

Sequence	Text That Is Substituted
\1, \2, ... \9	The value matched by the *n*th grouped subexpression
\&	The last match
\`	The part of the string before the match
\'	The part of the string after the match
\+	The highest-numbered group matched
\k<*name*>	The named capture

hex

str.hex → *int*

Treats leading characters from *str* as a string of hexadecimal digits (with an optional sign and an optional 0x) and returns the corresponding number. Zero is returned on error.

```
"0x0a".hex     # =>   10
"-1234".hex    # =>   -4660
"0".hex        # =>   0
"wombat".hex   # =>   0
```

include?

str.include?(*string*) → true or false

1.9

Returns true if *str* contains the given string.

```
"hello".include? "lo"   # =>   true
"hello".include? "ol"   # =>   false
"hello".include? ?h     # =>   true
```

index

str.index(*string* ⟨ , *offset* ⟩)
str.index(*regexp* ⟨ , *offset* ⟩) → *int* or nil

Returns the index of the first occurrence of the given substring or pattern in *str*. Returns nil if not found. If the second parameter is present, it specifies the position in the string to begin the search.

```
"hello".index('e')             # =>   1
"hello".index('lo')            # =>   3
"hello".index('a')             # =>   nil
"hello".index(/[aeiou]/, -3)   # =>   4
```

insert

str.insert(*index*, *string*) → *str*

Inserts *string* before the character at the given *index*, modifying *str*. Negative indices count from the end of the string and insert *after* the given character. After the insertion, *str* will contain *string* starting at *index*.

```
"abcd".insert(0, 'X')    # =>   "Xabcd"
"abcd".insert(3, 'X')    # =>   "abcXd"
"abcd".insert(4, 'X')    # =>   "abcdX"
"abcd".insert(-3, 'X')   # =>   "abXcd"
"abcd".insert(-1, 'X')   # =>   "abcdX"
```

intern

str.intern → *symbol*

Returns the Symbol corresponding to *str*, creating the symbol if it did not previously exist. Can intern any string, not just identifiers. See Symbol#id2name on page 703.

```
"Koala".intern   # =>   :Koala
sym = "$1.50 for a soda!?!?".intern
sym.to_s         # =>   "$1.50 for a soda!?!?"
```

length

str.length → *int*

Returns the number of characters in *str*. See also String#bytesize.

lines

str.lines(*sep*=$/) → *enum*
str.lines(*sep*=$/) {| *substr* | *block* } → *str*

1.9

Splits *str* using the supplied parameter as the record separator ($/ by default), passing each substring in turn to the supplied block. If a zero-length record separator is supplied, the string is split into paragraphs, each terminated by multiple \n characters. With no block, returns a enumerator.

```
print "Example one\n"
"hello\nworld".lines {|s| p s}
print "Example two\n"
"hello\nworld".lines('l') {|s| p s}
print "Example three\n"
"hello\n\n\nworld".lines('') {|s| p s}
```

produces:

```
Example one
"hello\n"
"world"
Example two
"hel"
"l"
"o\nworl"
"d"
Example three
"hello\n\n\n"
"world"
```

ljust

str.ljust(*width*, *padding*=" ") → *string*

If *width* is greater than the length of *str*, returns a new String of length *width* with *str* left justified and padded with copies of *padding*; otherwise, returns a copy of *str*.

```
"hello".ljust(4)             # =>   "hello"
"hello".ljust(20)            # =>   "hello␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
"hello".ljust(20, "*")       # =>   "hello***************"
"hello".ljust(20, " dolly")  # =>   "hello␣dolly␣dolly␣do"
```

lstrip

str.lstrip → *string*

Returns a copy of *str* with leading whitespace characters removed. Also see the methods String#rstrip and String#strip.

```
"  hello  ".lstrip      # =>  "hello␣␣"
"\000 hello  ".lstrip   # =>  "\x00␣hello␣␣"
"hello".lstrip          # =>  "hello"
```

lstrip!

str.lstrip! → self or nil

Removes leading whitespace characters from *str*, returning nil if no change was made. See also String#rstrip! and String#strip!.

```
"  hello  ".lstrip!   # =>  "hello␣␣"
"hello".lstrip!       # =>  nil
```

match

str.match(*pattern*) → *match_data* or nil
str.match(*pattern*) {| *matchdata* | *block* } → *obj*

1.9

Converts *pattern* to a Regexp (if it isn't already one) and then invokes its match method on *str*. If a block is given, the block is passed the MatchData object, and the match method returns the value of the block.

```
'seed'.match('(.)\1')                    # =>  #<MatchData "ee" 1:"e">
'seed'.match('(.)\1')[0]                 # =>  "ee"
'seed'.match(/(.)\1/)[0]                 # =>  "ee"
'seed'.match('ll')                       # =>  nil
'seed'.match('ll') {|md| md[0].upcase }  # =>  nil
'seed'.match('xx')                       # =>  nil
```

next

str.next → *string*

Synonym for String#succ.

next!

str.next! → *str*

Synonym for String#succ!.

oct

str.oct → *int*

Treats leading characters of *str* as a string of octal digits (with an optional sign) and returns the corresponding number. Returns 0 if the conversion fails.

```
"123".oct       # =>  83
"-377".oct      # =>  -255
"bad".oct       # =>  0
"0377bad".oct   # =>  255
```

ord

str.ord → *int*

1.9

Returns the integer code point of the first character of *str*. Note that it isn't quite the inverse of Integer#chr, because the latter does not deal with encodings.

```
# encoding: utf-8
"d".ord     # =>   100
"dog".ord   # =>   100
"δ".ord     # =>   8706
```

partition

str.partition(*pattern*) → [*before*, *match*, *after*]

1.9

Searches *str* for *pattern* (which may be a string or a regular expression). Returns a three-element array containing the part of the string before the pattern, the part that matched the pattern, and the part after the match. If the pattern does not match, the entire string will be returned as the first element of the array, and the other two entries will be empty strings.

```
"THX1138".partition("11")     # =>   ["THX", "11", "38"]
"THX1138".partition(/\d\d/)   # =>   ["THX", "11", "38"]
"THX1138".partition("99")     # =>   ["THX1138", "", ""]
```

replace

str.replace(*string*) → *str*

Replaces the contents, encoding, and taintedness of *str* with the corresponding values in *string*.

```
s = "hello"         # =>   "hello"
s.replace "world"   # =>   "world"
```

reverse

str.reverse → *string*

Returns a new string with the characters from *str* in reverse order.

```
# Every problem contains its own solution...
"stressed".reverse   # =>   "desserts"
```

reverse!

str.reverse! → *str*

Reverses *str* in place.

rindex

str.rindex(*string* ⟨ , *int* ⟩) → *int* or nil
str.rindex(*regexp* ⟨ , *int* ⟩) → *int* or nil

Returns the index of the last occurrence of the given substring, character, or pattern in *str*. Returns nil if not found. If the second parameter is present, it specifies the position in the string to end the search—characters beyond this point will not be considered.

```
"hello".rindex('e')             # =>   1
"hello".rindex('l')             # =>   3
"hello".rindex('a')             # =>   nil
"hello".rindex(/[aeiou]/, -2)   # =>   1
```

rjust

str.rjust(*width*, *padding*=" ") → *string*

If *width* is greater than the length of *str*, returns a new String of length *width* with *str* right justified and padded with copies of *padding*; otherwise, returns a copy of *str*.

```
"hello".rjust(4)              # =>  "hello"
"hello".rjust(20)             # =>  "              hello"
"hello".rjust(20, "-")        # =>  "---------------hello"
"hello".rjust(20, "padding")  # =>  "paddingpaddingphello"
```

rpartition

str.rpartition(*pattern*) → [*before*, *match after*]

1.9

Searches *str* for *pattern* (which may be a string or a regular expression), starting at the end of the string. Returns a three-element array containing the part of the string before the pattern, the part that matched the pattern, and the part after the match. If the pattern does not match, the entire string will be returned as the last element of the array, and the other two entries will be empty strings.

```
"THX1138".rpartition("1")    # =>  ["THX1", "1", "38"]
"THX1138".rpartition(/1\d/)  # =>  ["THX1", "13", "8"]
"THX1138".rpartition("99")   # =>  ["", "", "THX1138"]
```

rstrip

str.rstrip → *string*

Returns a copy of *str*, stripping first trailing NUL characters and then stripping trailing whitespace characters. See also String#lstrip and String#strip.

```
"  hello  ".rstrip       # =>  "  hello"
"  hello \000 ".rstrip   # =>  "  hello"
"  hello  \000".rstrip   # =>  "  hello"
"hello".rstrip           # =>  "hello"
```

rstrip!

str.rstrip! → self or nil

Removes trailing NUL characters and then removes trailing whitespace characters from *str*. Returns nil if no change was made. See also String#lstrip! and #strip!.

```
"  hello  ".rstrip!   # =>  "  hello"
"hello".rstrip!       # =>  nil
```

scan

str.scan(*pattern*) → *array*
str.scan(*pattern*) {| *match*, ... | *block* } → *str*

Both forms iterate through *str*, matching the pattern (which may be a Regexp or a String). For each match, a result is generated and either added to the result array or passed to the block. If the pattern contains no groups, each individual result consists of the matched string, $&. If the pattern contains groups, each individual result is itself an array containing one entry per group. If the pattern is a String, it is interpreted literally (in other words, it is not taken to be a regular expression pattern).

```
a = "cruel world"
a.scan(/\w+/)        # =>  ["cruel", "world"]
a.scan(/.../)        # =>  ["cru", "el ", "wor"]
a.scan(/(...)/)      # =>  [["cru"], ["el "], ["wor"]]
a.scan(/(..)(..)/)   # =>  [["cr", "ue"], ["l ", "wo"]]
```

And the block form:

```
a.scan(/\w+/) {|w| print "<<#{w}>> " }
puts
a.scan(/(.)(.)/) {|a,b| print b, a }
puts
```

produces:

```
<<cruel>> <<world>>
rceu lowlr
```

setbyte

str.setbyte(*offset*, *byte*) → *byte*

1.9

Sets the byte at *offset* (starting from the end of the string if the offset is negative) to *byte*. Cannot be used to change the length of the string. Does not change the encoding of the string.

```
str = "defog"
# a utf-8 delta character
str.setbyte(0, 226)   # =>   226
str.setbyte(1, 136)   # =>   136
str.setbyte(2, 130)   # =>   130
str                   # =>   "\xE2\x88\x82og"
str.length            # =>   5
str.force_encoding("utf-8")
str.length            # =>   3
str                   # =>   "δog"
```

size

str.size → *int*

Synonym for String#length.

slice

str.slice(*int*) → *string* or nil
str.slice(*int*, *int*) → *string* or nil
str.slice(*range*) → *string* or nil
str.slice(*regexp*) → *string* or nil
str.slice(*match_string*) → *string* or nil

Synonym for String#[ ].

```
a = "hello there"
a.slice(1)        # =>   "e"
a.slice(1,3)      # =>   "ell"
a.slice(1..3)     # =>   "ell"
a.slice(-3,2)     # =>   "er"
a.slice(-4..-2)   # =>   "her"

a.slice(-2..-4)        # =>   ""
a.slice(/th[aeiou]/)   # =>   "the"
a.slice("lo")          # =>   "lo"
a.slice("bye")         # =>   nil
```

slice!

str.slice!(*int*) → *string* or nil
str.slice!(*int*, *int*) → *string* or nil
str.slice!(*range*) → *string* or nil
str.slice!(*regexp*) → *string* or nil
str.slice!(*match_string*) → *string* or nil

Deletes the specified portion from *str*and returns the portion deleted. The forms that take a Fixnum will raise an IndexError if the value is out of range; the Range form will raise a RangeError, and the Regexp and String forms will silently not change the string.

```
string = "this is a string"
string.slice!(2)        # =>   "i"
string.slice!(3..6)     # =>   " is "
string.slice!(/s.*t/)   # =>   "sa st"
string.slice!("r")      # =>   "r"
string                  # =>   "thing"
```

split

str.split(*pattern*=$;, ⟨ *limit* ⟩) → *array*

Divides *str* into substrings based on a delimiter, returning an array of these substrings.

If *pattern* is a String, then its contents are used as the delimiter when splitting *str*. If *pattern* is a single space, *str* is split on whitespace, with leading whitespace and runs of contiguous whitespace characters ignored.

If *pattern* is a Regexp, *str* is divided where the pattern matches. Whenever the pattern matches a zero-length string, *str* is split into individual characters. If pattern includes groups, these groups will be included in the returned values.

If *pattern* is omitted, the value of $; is used. If $; is nil (which is the default), *str* is split on whitespace as if "␣" were specified.

If the *limit* parameter is omitted, trailing empty fields are suppressed. If *limit* is a positive number, at most that number of fields will be returned (if *limit* is 1, the entire string is returned as the only entry in an array). If negative, there is no limit to the number of fields returned, and trailing null fields are not suppressed.

```
" now's   the time".split        # =>   ["now's", "the", "time"]
" now's   the time".split(' ')   # =>   ["now's", "the", "time"]
" now's   the time".split(/ /)   # =>   ["", "now's", "", "", "the",
                                       "time"]
"a@1bb@2ccc".split(/@\d/)        # =>   ["a", "bb", "ccc"]
"a@1bb@2ccc".split(/@(\d)/)      # =>   ["a", "1", "bb", "2", "ccc"]
"1, 2.34,56, 7".split(/,\s*/)    # =>   ["1", "2.34", "56", "7"]
"hello".split(//)                # =>   ["h", "e", "l", "l", "o"]
"hello".split(//, 3)             # =>   ["h", "e", "llo"]
"hi mom".split(/\s*/)            # =>   ["h", "i", "m", "o", "m"]

"".split                         # =>   []
```

```
"mellow yellow".split("ello")   # =>   ["m", "w y", "w"]
"1,2,,3,4,,".split(',')         # =>   ["1", "2", "", "3", "4"]
"1,2,,3,4,,".split(',', 4)      # =>   ["1", "2", "", "3,4,,"]
"1,2,,3,4,,".split(',', -4)     # =>   ["1", "2", "", "3", "4", "", ""]
```

squeeze

str.squeeze(⟨ *string* ⟩*) → *squeezed_string*

Builds a set of characters from the *string* parameter(s) using the procedure described for String#count on page 677. Returns a new string where runs of the same character that occur in this set are replaced by a single character. If no arguments are given, all runs of identical characters are replaced by a single character.

```
"yellow moon".squeeze                  # =>   "yelow mon"
"  now   is  the".squeeze(" ")         # =>   " now is the"
"putters putt balls".squeeze("m-z")    # =>   "puters put balls"
```

squeeze!

str.squeeze!(⟨ *string* ⟩*) → *str* or nil

Squeezes *str* in place, returning *str*. Returns nil if no changes were made.

start_with?

str.start_with?(⟨ suffix ⟩+) → true or false

1.9

Returns true if *str* starts with any of the given prefixes.

```
"Apache".start_with?("Apa")                          # =>   true
"ruby code".start_with?("python", "perl", "ruby")    # =>   true
```

strip

str.strip → *string*

Returns a copy of *str* with leading whitespace and trailing NUL and whitespace characters removed.

```
"    hello    ".strip   # =>   "hello"
"\tgoodbye\r\n".strip   # =>   "goodbye"
"goodbye  \000".strip   # =>   "goodbye"
"goodbye \000 ".strip   # =>   "goodbye"
```

strip!

str.strip! → *str* or nil

Removes leading whitespace and trailing NUL and whitespace characters from *str*. Returns nil if *str* was not altered.

sub

str.sub(*pattern*, *replacement*) → *string*
str.sub(*pattern*) {| *match* | *block* } → *string*

Returns a copy of *str* with the *first* occurrence of *pattern* replaced with either *replacement* or the value of the block. See the description of String#gsub on page 681 for a description of the parameters.

```
"hello".sub(/[aeiou]/, '*')                  # =>   "h*llo"
"hello".sub(/([aeiou])/, '<\1>')             # =>   "h<e>llo"
"hello".sub(/./) {|s| s[0].to_s + ' '}       # =>   "h ello"
"hello".sub(/(?<double>l)/, '-\k<double>-')  # =>   "he-l-lo"
```

sub! *str*.sub!(*pattern*, *replacement*) → *str* or nil
str.sub!(*pattern*) {| *match* | *block* } → *str* or nil

Performs the substitutions of String#sub in place, returning *str*. Returns nil if no substitutions were performed.

succ *str*.succ → *string*

Returns the successor to *str*. The successor is calculated by incrementing characters starting from the rightmost alphanumeric (or the rightmost character if there are no alphanumerics) in the string. Incrementing a digit always results in another digit, and incrementing a letter results in another letter of the same case. Incrementing nonalphanumerics uses the underlying character set's collating sequence.

1.9

If the increment generates a "carry," the character to the left of it is incremented. This process repeats until there is no carry, adding a character if necessary. An exception is when the carry is generated by a sequence of digits in a string containing digits, nonalpha characters, and more digits, in which case the carry applies to the digits. This allows for incrementing (for example) numbers with decimal places.

```
"abcd".succ        # =>   "abce"
"THX1138".succ     # =>   "THX1139"
"<<koala>>".succ   # =>   "<<koalb>>"
"1999zzz".succ     # =>   "2000aaa"
"ZZZ9999".succ     # =>   "AAAA0000"
"***".succ         # =>   "**+"
"1.9".succ         # =>   "2.0"
"1//9".succ        # =>   "2//0"
"1/9/9/9".succ     # =>   "2/0/0/0"
"1x9".succ         # =>   "1y0"
```

succ! *str*.succ! → *str*

Equivalent to String#succ but modifies the receiver in place.

sum *str*.sum(*n*=16) → *int*

Returns a basic *n*-bit checksum of the characters in *str*, where *n* is the optional parameter, defaulting to 16. The result is simply the sum of the binary value of each character in *str* modulo $2^n - 1$. This is not a particularly good checksum—see the digest libraries on page 745 for better alternatives.

```
"now is the time".sum      # =>   1408
"now is the time".sum(8)   # =>   128
```

swapcase *str*.swapcase → *string*

Returns a copy of *str* with uppercase alphabetic characters converted to lowercase and lowercase characters converted to uppercase. The mapping depends on the string encoding, but not all encodings produce expected results.

```
# encoding: utf-8
"Hello".swapcase          # =>   "hELLO"
"cYbEr_PuNk11".swapcase   # =>   "CyBeR_pUnK11"
"δ0g".swapcase            # =>   "δoG"
```

swapcase!

str.swapcase! → *str* or nil

Equivalent to String#swapcase but modifies *str* in place, returning *str*. Returns nil if no changes were made.

to_c

str.to_c → *float*

Returns the result of interpreting leading characters in *str* as a complex number. Extraneous characters past the end of a valid number are ignored. If there is not a valid number at the start of *str*, Complex(0,0) is returned. The method never raises an exception.

```
"123".to_c       # =>   123+0i
"4+5/6i".to_c    # =>   4+5/6i
"thx1138".to_c   # =>   0+0i
```

to_f

str.to_f → *complex*

Returns the result of interpreting leading characters in *str* as a floating-point number. Extraneous characters past the end of a valid number are ignored. If there is not a valid number at the start of *str*, 0.0 is returned. The method never raises an exception (use Kernel.Float to validate numbers).

```
"123.45e1".to_f        # =>   1234.5
"45.67 degrees".to_f   # =>   45.67
"thx1138".to_f         # =>   0.0
```

to_i

str.to_i(*base*=10) → *int*

Returns the result of interpreting leading characters in *str* as an integer base *base* (2 to 36). Given a base of zero, to_i looks for leading 0, 0b, 0o, 0d, or 0x and sets the base accordingly. Leading spaces are ignored, and leading plus or minus signs are honored. Extraneous characters past the end of a valid number are ignored. If there is not a valid number at the start of *str*, 0 is returned. The method never raises an exception.

```
"12345".to_i             # =>   12345
"99 red balloons".to_i   # =>   99
"0a".to_i                # =>   0
"0a".to_i(16)            # =>   10
"0x10".to_i              # =>   0
"0x10".to_i(0)           # =>   16
"-0x10".to_i(0)          # =>   -16
"hello".to_i             # =>   0
"hello".to_i(30)         # =>   14167554
"1100101".to_i(2)        # =>   101
"1100101".to_i(8)        # =>   294977
"1100101".to_i(10)       # =>   1100101
"1100101".to_i(16)       # =>   17826049
"1100101".to_i(24)       # =>   199066177
```

to_r *str*.to_r → *float*

1.9

Returns the result of interpreting leading characters in *str* as a rational number. Extraneous characters past the end of a valid number are ignored. If there is not a valid number at the start of *str*, Rational(0,1) is returned. The method never raises an exception.

```
"123".to_r       # =>   123/1
"5/6".to_r       # =>   5/6
"25/100".to_r    # =>   1/4
"thx1138".to_r   # =>   (0/1)
```

to_s *str*.to_s → *str*

Returns the receiver.

to_str *str*.to_str → *str*

Synonym for String#to_s. to_str is used by methods such as String#concat to convert their arguments to a string. Unlike to_s, which is supported by almost all classes, to_str is normally implemented only by those classes that act like strings. Of the built-in classes, only Exception and String implement to_str.

to_sym *str*.to_s → *symbol*

Returns the symbol for *str*. This can create symbols that cannot be represented using the :xxx notation. A synonym for String#intern.

```
s = 'cat'.to_sym       # =>   :cat
s == :cat              # =>   true
'cat and dog'.to_sym   # =>   :"cat and dog"
s == :'cat and dog'    # =>   false
```

tr *str*.tr(*from_string*, *to_string*) → *string*

Returns a copy of *str* with the characters in *from_string* replaced by the corresponding characters in *to_string*. If *to_string* is shorter than *from_string*, it is padded with its last character. Both strings may use the c_1–c_2 notation to denote ranges of characters, and *from_string* may start with a ^, which denotes all characters except those listed.

```
"hello".tr('aeiou', '*')    # =>   "h*ll*"
"hello".tr('^aeiou', '*')   # =>   "*e**o"
"hello".tr('el', 'ip')      # =>   "hippo"
"hello".tr('a-y', 'b-z')    # =>   "ifmmp"
```

tr! *str*.tr!(*from_string*, *to_string*) → *str* or nil

Translates *str* in place, using the same rules as String#tr. Returns *str* or returns nil if no changes were made.

tr_s *str*.tr_s(*from_string*, *to_string*) → *string*

Processes a copy of *str* as described under String#tr and then removes duplicate characters in regions that were affected by the translation.

```
"hello".tr_s('l', 'r')     # =>   "hero"
"hello".tr_s('el', '*')    # =>   "h*o"
"hello".tr_s('el', 'hx')   # =>   "hhxo"
```

tr_s! *str*.tr_s!(*from_string*, *to_string*) → *str* or nil

Performs String#tr_s processing on *str* in place, returning *str*. Returns nil if no changes were made.

unpack *str*.unpack(*format*) → *array*

Decodes *str* (which may contain binary data) according to the format string, returning an array of the extracted values. The format string consists of a sequence of single-character directives, summarized in Table 27.17 on the next page. Each directive may be followed by a number, indicating the number of times to repeat this directive. An asterisk (*) will use up all remaining elements. The directives sSiIlL may each be followed by an underscore (_) or bang (!) to use the underlying platform's native size for the specified type; otherwise, it uses a platform-independent consistent size. Spaces are ignored in the format string. Comments starting with # to the next newline or end of string are also ignored. The encoding of the string is ignored; unpack treats the string as a sequence of bytes. See also Array#pack on page 457.

1.9

1.9

```
"abc \0\0abc \0\0".unpack('A6Z6')   # =>   ["abc", "abc "]
"abc \0\0".unpack('a3a3')           # =>   ["abc", " \x00\x00"]
"aa".unpack('b8B8')                 # =>   ["10000110", "01100001"]
"aaa".unpack('h2H2c')               # =>   ["16", "61", 97]
"\xfe\xff\xfe\xff".unpack('sS')     # =>   [-2, 65534]
"now=20is".unpack('M*')             # =>   ["now is"]
"whole".unpack('xax2aX2aX1aX2a')    # =>   ["h", "e", "l", "l", "o"]
```

upcase *str*.upcase → *string*

Returns a copy of *str* with all lowercase letters replaced with their uppercase counterparts. The mapping depends on the string encoding, but not all encodings produce expected results.

```
# encoding: utf-8
"hEllO".upcase   # =>   "HELLO"
"δog".upcase     # =>   "δOG"
```

upcase! *str*.upcase! → *str* or nil

Upcases the contents of *str*, returning nil if no changes were made.

Table 27.17. Directives for String#unpack

	Format	Function	Returns
1.9	A	Sequence of bytes with trailing NULs and ASCII spaces removed.	String
1.9	a	Sequence of bytes.	String
	B	Extracts bits from each byte (MSB first).	String
	b	Extracts bits from each byte (LSB first).	String
1.9	C	Extracts a byte as an unsigned integer.	Fixnum
1.9	c	Extracts a byte as an integer.	Fixnum
	d,D	Treat *sizeof(double)* bytes as a native double.	Float
	E	Treats *sizeof(double)* bytes as a double in little-endian byte order.	Float
	e	Treats *sizeof(float)* bytes as a float in little-endian byte order.	Float
	f,F	Treats *sizeof(float)* bytes as a native float.	Float
	G	Treats *sizeof(double)* bytes as a double in network byte order.	Float
	g	Treats *sizeof(float)* bytes as a float in network byte order.	Float
	H	Extracts hex nibbles from each byte (most significant first).	String
	h	Extracts hex nibbles from each byte (least significant first).	String
	I	Treats *sizeof(int)*[1] successive bytes as an unsigned native integer.	Integer
	i	Treats *sizeof(int)*[1] successive bytes as a signed native integer.	Integer
	L	Treats four[1] successive bytes as an unsigned native long integer.	Integer
	l	Treats four[1] successive characters as a signed native long integer.	Integer
	M	Extracts a quoted-printable string.	String
	m	Extracts a Base64-encoded string. By default, accepts \n and \r. "m0" rejects these.	String
	N	Treats four bytes as an unsigned long in network byte order.	Fixnum
	n	Treats two bytes as an unsigned short in network byte order.	Fixnum
	P	Treats *sizeof(char *)* bytes as a pointer and returns *len* bytes from the referenced location.	String
	p	Treats *sizeof(char *)* bytes as a pointer to a null-terminated string.	String
	Q	Treats eight bytes as an unsigned quad word (64 bits).	Integer
	q	Treats eight bytes as a signed quad word (64 bits).	Integer
	S	Treats two[1] bytes characters as an unsigned short in native byte order.	Fixnum
	s	Treats two[1] successive bytes as a signed short in native byte order.	Fixnum
	U	Extracts UTF-8 characters as unsigned integers.	Integer
	u	Extracts a UU-encoded string.	String
	V	Treats four bytes as an unsigned long in little-endian byte order.	Fixnum
	v	Treats two bytes as an unsigned short in little-endian byte order.	Fixnum
	w	BER-compressed integer (see Array#pack for more information).	Integer
	X	Skips backward one byte.	—
	x	Skips forward one byte.	—
	Z	String with trailing NULs removed.	String
	@	Skips to the byte offset given by the length argument.	—

1.9 [1] May be modified by appending _ or ! to the directive.

upto

str.upto(*string*) {| *s* | *block* } → *str* or *enumerator*

1.9

Iterates through successive values, starting at *str* and ending at *string* inclusive, passing each value in turn to the block. The String#succ method is used to generate each value. Returns an Enumerator object if no block is given.

```
"a8".upto("b6") {|s| print s, ' ' }
for s in "a8".."b6"
  print s, ' '
end
```

produces:

```
a8 a9 b0 b1 b2 b3 b4 b5 b6
a8 a9 b0 b1 b2 b3 b4 b5 b6
```

valid_encoding?

str.valid_encoding? → true or false

1.9

Returns true if *str* contains a valid byte sequence in its current encoding.

```
# encoding: binary
str = "\xE2"
str.force_encoding("utf-8")
str.valid_encoding?  # =>  false
str = "\xE2\x88\x82"
str.force_encoding("utf-8")
str.valid_encoding?  # =>  true
```