Home

Pages Classes Methods

Search

Table of Contents

Show/hide navigation

Parent

Object

Included Modules

Comparable

Methods

class String

A String object has an arbitrary sequence of bytes, typically representing text or binary data. A String object may be created using `String::new` or as literals.

`String` objects differ from `Symbol` objects in that `Symbol` objects are designed to be used as identifiers, instead of text or data.

You can create a String object explicitly with:

- string literal.
- heredoc literal.

You can convert certain objects to Strings with:

- Method `String`.

Some String methods modify `self`. Typically, a method whose name ends with `!` modifies `self` and returns `self`; often a similarly named method (without the `!`) returns a new string.

In general, if there exist both bang and non-bang version of method, the bang! mutates and the non-bang! does not. However, a method without a bang can also mutate, such as `String#replace`.

Substitution Methods

These methods perform substitutions:

- `String#sub`: One substitution (or none); returns a new string.
- `String#sub!`: One substitution (or none); returns `self`.
- `String#gsub`: Zero or more substitutions; returns a new string.
- `String#gsub!`: Zero or more substitutions; returns `self`.

Each of these methods takes:

- A first argument, `pattern` (string or regexp), that specifies the substring(s) to be replaced.

- Either of these:
  - A second argument, `replacement` (string or hash), that determines the replacing string.
  - A block that will determine the replacing string.

The examples in this section mostly use methods `String#sub` and `String#gsub`; the principles illustrated apply to all four substitution methods.

Argument `pattern`

Argument `pattern` is commonly a regular expression:

```
s = 'hello'
s.sub(/[aeiou]/, '*')# => "h*llo"
s.gsub(/[aeiou]/, '*') # => "h*ll*"
s.gsub(/[aeiou]/, '')# => "hll"
s.sub(/ell/, 'al')   # => "halo"
s.gsub(/xyzzy/, '*') # => "hello"
'THX1138'.gsub(/\d+/, '00') # => "THX00"
```

When `pattern` is a string, all its characters are treated as ordinary characters (not as regexp special characters):

```
'THX1138'.gsub('\d+', '00') # => "THX1138"
```

String `replacement`

If `replacement` is a string, that string will determine the replacing string that is to be substituted for the matched text.

Each of the examples above uses a simple string as the replacing string.

String `replacement` may contain back-references to the pattern's captures:

- `\n` (*n* a non-negative integer) refers to `$n`.
- `\k<name>` refers to the named capture `name`.

See regexp.rdoc for details.

Note that within the string `replacement`, a character combination such as `$&` is treated as ordinary text, and not as a special match variable. However, you may refer to some special match variables using these combinations:

- `\&` and `\0` correspond to `$&`, which contains the complete matched text.
- `\'` corresponds to `$'`, which contains string after match.
- `\`` corresponds to `$``, which contains string before match.
- `\+` corresponds to `$+`, which contains last capture group.

See regexp.rdoc for details.

Note that `\\` is interpreted as an escape, i.e., a single backslash.

Note also that a string literal consumes backslashes. See String Literals for details about string literals.

A back-reference is typically preceded by an additional backslash. For example, if you want to write a back-reference `\&` in `replacement` with a double-quoted string literal, you need to write `"..\\&.."`.

If you want to write a non-back-reference string `\&` in `replacement`, you need first to escape the backslash to prevent this method from interpreting it as a back-reference, and then you need to escape the backslashes again to prevent a string literal from consuming them: `"..\\\\&.."`.

You may want to use the block form to avoid a lot of backslashes.

Hash `replacement`

If argument `replacement` is a hash, and `pattern` matches one of its keys, the replacing string is the value for that key:

```
h = {'foo' => 'bar', 'baz' => 'bat'}
'food'.sub('foo', h) # => "bard"
```

Note that a symbol key does not match:

```
h = {foo: 'bar', baz: 'bat'}
'food'.sub('foo', h) # => "d"
```

Block

In the block form, the current match string is passed to the block; the block's return value becomes the replacing string:

```
 s = '@'
'1234'.gsub(/\d/) {|match| s.succ! } # => "ABCD"
```

Special match variables such as `$1`, `$2`, `$`` , `$&`, and `$'` are set appropriately.

Whitespace in Strings

In class String, *whitespace* is defined as a contiguous sequence of characters consisting of any mixture of the following:

- NL (null): `"\x00"`, `"\u0000"`.
- HT (horizontal tab): `"\x09"`, `"\t"`.
- LF (line feed): `"\x0a"`, `"\n"`.
- VT (vertical tab): `"\x0b"`, `"\v"`.
- FF (form feed): `"\x0c"`, `"\f"`.

- CR (carriage return): `"\x0d"`, `"\r"`.
- SP (space): `"\x20"`, `" "`.

Whitespace is relevant for these methods:

- `lstrip`, `lstrip!`: strip leading whitespace.
- `rstrip`, `rstrip!`: strip trailing whitespace.
- `strip`, `strip!`: strip leading and trailing whitespace.

String Slices

A *slice* of a string is a substring that is selected by certain criteria.

These instance methods make use of slicing:

- `String#[]` (also aliased as `String#slice`) returns a slice copied from `self`.
- `String#[]=` returns a copy of `self` with a slice replaced.
- `String#slice!` returns `self` with a slice removed.

Each of the above methods takes arguments that determine the slice to be copied or replaced.

The arguments have several forms. For string `string`, the forms are:

- `string[index]`.
- `string[start, length]`.
- `string[range]`.
- `string[regexp, capture = 0]`.
- `string[substring]`.

`string[index]`

When non-negative integer argument `index` is given, the slice is the 1-character substring found in `self` at character offset `index`:

```
'bar'[0]       # => "b"
'bar'[2]       # => "r"
'bar'[20]      # => nil
'тест'[2]      # => "с"
'こんにちは'[4]  # => "は"
```

When negative integer `index` is given, the slice begins at the offset given by counting backward from the end of `self`:

```
'bar'[-3]          # => "b"
'bar'[-1]          # => "r"
'bar'[-20]         # => nil
```

`string[start, length]`

When non-negative integer arguments `start` and `length` are given, the slice begins at character offset `start`, if it exists, and continues for `length` characters, if available:

```
'foo'[0, 2]        # => "fo"
'тест'[1, 2]       # => "ес"
'こんにちは'[2, 2]  # => "にち"
# Zero length.
'foo'[2, 0]        # => ""
# Length not entirely available.
'foo'[1, 200]      # => "oo"
# Start out of range.
'foo'[4, 2]       # => nil
```

Special case: if `start` is equal to the length of `self`, the slice is a new empty string:

```
'foo'[3, 2]   # => ""
'foo'[3, 200] # => ""
```

When negative `start` and non-negative `length` are given, the slice beginning is determined by counting backward from the end of `self`, and the slice continues for `length` characters, if available:

```
'foo'[-2, 2]    # => "oo"
'foo'[-2, 200]  # => "oo"
# Start out of range.
'foo'[-4, 2]     # => nil
```

When negative `length` is given, there is no slice:

```
'foo'[1, -1]  # => nil
'foo'[-2, -1] # => nil
```

`string[range]`

When Range argument `range` is given, creates a substring of `string` using the indices in `range`. The slice is then determined as above:

```
'foo'[0..1]    # => "fo"
'foo'[0, 2]    # => "fo"

'foo'[2...2]   # => ""
'foo'[2, 0]    # => ""

'foo'[1..200]  # => "oo"
'foo'[1, 200]  # => "oo"
```

```
'foo'[4..5]    # => nil
'foo'[4, 2]    # => nil

'foo'[-4..-3]  # => nil
'foo'[-4, 2]   # => nil

'foo'[3..4]    # => ""
'foo'[3, 2]    # => ""

'foo'[-2..-1]  # => "oo"
'foo'[-2, 2]   # => "oo"

'foo'[-2..197] # => "oo"
'foo'[-2, 200] # => "oo"
```

`string[regexp, capture = 0]`

When the Regexp argument `regexp` is given, and the `capture` argument is `0`, the slice is the first matching substring found in `self`:

```
'foo'[/o/] # => "o"
'foo'[/x/] # => nil
s = 'hello there'
s[/[aeiou](.)\1/] # => "ell"
s[/[aeiou](.)\1/, 0] # => "ell"
```

If argument `capture` is given and not `0`, it should be either an capture group index (integer) or a capture group name (string or symbol); the slice is the specified capture (see Groups and Captures at Regexp):

```
s = 'hello there'
s[/[aeiou](.)\1/, 1] # => "l"
s[/(?<vowel>[aeiou])(?<non_vowel>[^aeiou])/, "non_vowel"] # => "l"
s[/(?<vowel>[aeiou])(?<non_vowel>[^aeiou])/, :vowel] # => "e"
```

If an invalid capture group index is given, there is no slice. If an invalid capture group name is given, `IndexError` is raised.

`string[substring]`

When the single String argument `substring` is given, returns the substring from `self` if found, otherwise `nil`:

```
'foo'['oo'] # => "oo"
'foo'['xx'] # => nil
```

What's Here

First, what's elsewhere. Class String:

- Inherits from class Object.

- Includes module Comparable.

Here, class String provides methods that are useful for:

- Creating a String
- Frozen/Unfrozen Strings
- Querying
- Comparing
- Modifying a String
- Converting to New String
- Converting to Non-String
- Iterating

Methods for Creating a String

- `::new`: Returns a new string.
- `::try_convert`: Returns a new string created from a given object.

Methods for a Frozen/Unfrozen `String`

- `+@`: Returns a string that is not frozen: `self`, if not frozen; `self.dup` otherwise.
- `-@`: Returns a string that is frozen: `self`, if already frozen; `self.freeze` otherwise.
- `freeze`: Freezes `self`, if not already frozen; returns `self`.

Methods for Querying

Counts

- `length`, `size`: Returns the count of characters (not bytes).
- `empty?`: Returns `true` if `self.length` is zero; `false` otherwise.
- `bytesize`: Returns the count of bytes.
- `count`: Returns the count of substrings matching given strings.

Substrings

- #=~: Returns the index of the first substring that matches a given `Regexp` or other object; returns `nil` if no match is found.
- `index`: Returns the index of the *first* occurrence of a given substring; returns `nil` if none found.

- `rindex` : Returns the index of the *last* occurrence of a given substring; returns `nil` if none found.
- `include?` : Returns `true` if the string contains a given substring; `false` otherwise.
- `match` : Returns a `MatchData` object if the string matches a given `Regexp` ; `nil` otherwise.
- `match?` : Returns `true` if the string matches a given `Regexp` ; `false` otherwise.
- `start_with?` : Returns `true` if the string begins with any of the given substrings.
- `end_with?` : Returns `true` if the string ends with any of the given substrings.

Encodings

- `encoding` : Returns the `Encoding` object that represents the encoding of the string.
- `unicode_normalized?` : Returns `true` if the string is in Unicode normalized form; `false` otherwise.
- `valid_encoding?` : Returns `true` if the string contains only characters that are valid for its encoding.
- `ascii_only?` : Returns `true` if the string has only ASCII characters; `false` otherwise.

Other

- `sum` : Returns a basic checksum for the string: the sum of each byte.
- `hash` : Returns the integer hash code.

Methods for Comparing

- `==` , `===` : Returns `true` if a given other string has the same content as `self` .
- `eql?` : Returns `true` if the content is the same as the given other string.
- #<=>: Returns -1, 0, or 1 as a given other string is smaller than, equal to, or larger than `self` .
- `casecmp` : Ignoring case, returns -1, 0, or 1 as a given other string is smaller than, equal to, or larger than `self` .
- `casecmp?` : Returns `true` if the string is equal to a given string after Unicode case folding; `false` otherwise.

Methods for Modifying a String

Each of these methods modifies `self`.

Insertion

- `insert`: Returns `self` with a given string inserted at a given offset.
- `<<`: Returns `self` concatenated with a given string or integer.

Substitution

- `sub!`: Replaces the first substring that matches a given pattern with a given replacement string; returns `self` if any changes, `nil` otherwise.
- `gsub!`: Replaces each substring that matches a given pattern with a given replacement string; returns `self` if any changes, `nil` otherwise.
- `succ!`, `next!`: Returns `self` modified to become its own successor.
- `replace`: Returns `self` with its entire content replaced by a given string.
- `reverse!`: Returns `self` with its characters in reverse order.
- `setbyte`: Sets the byte at a given integer offset to a given value; returns the argument.
- `tr!`: Replaces specified characters in `self` with specified replacement characters; returns `self` if any changes, `nil` otherwise.
- `tr_s!`: Replaces specified characters in `self` with specified replacement characters, removing duplicates from the substrings that were modified; returns `self` if any changes, `nil` otherwise.

Casing

- `capitalize!`: Upcases the initial character and downcases all others; returns `self` if any changes, `nil` otherwise.
- `downcase!`: Downcases all characters; returns `self` if any changes, `nil` otherwise.
- `upcase!`: Upcases all characters; returns `self` if any changes, `nil` otherwise.
- `swapcase!`: Upcases each downcase character and downcases each upcase character; returns `self` if any changes, `nil` otherwise.

Encoding

- `encode!`: Returns `self` with all characters transcoded from one given encoding into another.
- `unicode_normalize!`: Unicode-normalizes `self`; returns `self`.
- `scrub!`: Replaces each invalid byte with a given character; returns `self`.
- `force_encoding`: Changes the encoding to a given encoding; returns `self`.

Deletion

- `clear` : Removes all content, so that `self` is empty; returns `self` .
- `slice!` , `[]=` : Removes a substring determined by a given index, start/length, range, regexp, or substring.
- `squeeze!` : Removes contiguous duplicate characters; returns `self` .
- `delete!` : Removes characters as determined by the intersection of substring arguments.
- `lstrip!` : Removes leading whitespace; returns `self` if any changes, `nil` otherwise.
- `rstrip!` : Removes trailing whitespace; returns `self` if any changes, `nil` otherwise.
- `strip!` : Removes leading and trailing whitespace; returns `self` if any changes, `nil` otherwise.
- `chomp!` : Removes trailing record separator, if found; returns `self` if any changes, `nil` otherwise.
- `chop!` : Removes trailing newline characters if found; otherwise removes the last character; returns `self` if any changes, `nil` otherwise.

Methods for Converting to New String

Each of these methods returns a new String based on `self` , often just a modified copy of `self` .

Extension

- `*` : Returns the concatenation of multiple copies of `self` ,
- `+` : Returns the concatenation of `self` and a given other string.
- `center` : Returns a copy of `self` centered between pad substring.
- `concat` : Returns the concatenation of `self` with given other strings.
- `prepend` : Returns the concatenation of a given other string with `self` .
- `ljust` : Returns a copy of `self` of a given length, right-padded with a given other string.
- `rjust` : Returns a copy of `self` of a given length, left-padded with a given other string.

Encoding

- `b` : Returns a copy of `self` with ASCII-8BIT encoding.
- `scrub` : Returns a copy of `self` with each invalid byte replaced with a given character.
- `unicode_normalize` : Returns a copy of `self` with each character Unicode-normalized.

- encode : Returns a copy of self with all characters transcoded from one given encoding into another.

Substitution

- dump : Returns a copy of self with all non-printing characters replaced by xHH notation and all special characters escaped.
- undump : Returns a copy of self with all \xNN notation replace by \uNNNN notation and all escaped characters unescaped.
- sub : Returns a copy of self with the first substring matching a given pattern replaced with a given replacement string;.
- gsub : Returns a copy of self with each substring that matches a given pattern replaced with a given replacement string.
- succ , next : Returns the string that is the successor to self .
- reverse : Returns a copy of self with its characters in reverse order.
- tr : Returns a copy of self with specified characters replaced with specified replacement characters.
- tr_s : Returns a copy of self with specified characters replaced with specified replacement characters, removing duplicates from the substrings that were modified.
- % : Returns the string resulting from formatting a given object into self

Casing

- capitalize : Returns a copy of self with the first character upcased and all other characters downcased.
- downcase : Returns a copy of self with all characters downcased.
- upcase : Returns a copy of self with all characters upcased.
- swapcase : Returns a copy of self with all upcase characters downcased and all downcase characters upcased.

Deletion

- delete : Returns a copy of self with characters removed
- delete_prefix : Returns a copy of self with a given prefix removed.
- delete_suffix : Returns a copy of self with a given suffix removed.
- lstrip : Returns a copy of self with leading whitespace removed.
- rstrip : Returns a copy of self with trailing whitespace removed.
- strip : Returns a copy of self with leading and trailing whitespace removed.
- chomp : Returns a copy of self with a trailing record separator removed, if found.

- chop : Returns a copy of self with trailing newline characters or the last character removed.
- squeeze : Returns a copy of self with contiguous duplicate characters removed.
- [] , slice : Returns a substring determined by a given index, start/length, or range, or string.
- byteslice : Returns a substring determined by a given index, start/length, or range.
- chr : Returns the first character.

Duplication

- to_s , $to_str: If self is a subclass of String, returns self copied into a String; otherwise, returns self .

Methods for Converting to Non-String

Each of these methods converts the contents of self to a non-String.

Characters, Bytes, and Clusters

- bytes : Returns an array of the bytes in self .
- chars : Returns an array of the characters in self .
- codepoints : Returns an array of the integer ordinals in self .
- getbyte : Returns an integer byte as determined by a given index.
- grapheme_clusters : Returns an array of the grapheme clusters in self .

Splitting

- lines : Returns an array of the lines in self , as determined by a given record separator.
- partition : Returns a 3-element array determined by the first substring that matches a given substring or regexp,
- rpartition : Returns a 3-element array determined by the last substring that matches a given substring or regexp,
- split : Returns an array of substrings determined by a given delimiter – regexp or string – or, if a block given, passes those substrings to the block.

Matching

- scan : Returns an array of substrings matching a given regexp or string, or, if a block given, passes each matching substring to the block.
- unpack : Returns an array of substrings extracted from self according to a given format.

- `unpack1` : Returns the first substring extracted from `self` according to a given format.

Numerics

- `hex` : Returns the integer value of the leading characters, interpreted as hexadecimal digits.
- `oct` : Returns the integer value of the leading characters, interpreted as octal digits.
- `ord` : Returns the integer ordinal of the first character in `self` .
- `to_i` : Returns the integer value of leading characters, interpreted as an integer.
- `to_f` : Returns the floating-point value of leading characters, interpreted as a floating-point number.

Strings and Symbols

- `inspect` : Returns copy of `self` , enclosed in double-quotes, with special characters escaped.
- `to_sym` , `intern` : Returns the symbol corresponding to `self` .

Methods for Iterating

- `each_byte` : Calls the given block with each successive byte in `self` .
- `each_char` : Calls the given block with each successive character in `self` .
- `each_codepoint` : Calls the given block with each successive integer codepoint in `self` .
- `each_grapheme_cluster` : Calls the given block with each successive grapheme cluster in `self` .
- `each_line` : Calls the given block with each successive line in `self` , as determined by a given record separator.
- `upto` : Calls the given block with each string value returned by successive calls to `succ` .

Public Class Methods

`new(string = '', **opts) → new_string`

Returns a new String that is a copy of `string` .

With no arguments, returns the empty string with the `Encoding` `ASCII-8BIT` :

```
s = String.new
s # => ""
s.encoding # => #<Encoding:ASCII-8BIT>
```

With optional argument `string` and no keyword arguments, returns a copy of `string` with the same encoding:

```
String.new('foo')               # => "foo"
String.new('тест')              # => "тест"
String.new('こんにちは')          # => "こんにちは"
```

(Unlike String.new, a string literal like `''` or a here document literal always has script encoding.)

With optional keyword argument `encoding`, returns a copy of `string` with the specified encoding; the `encoding` may be an `Encoding` object, an encoding name, or an encoding name alias:

```
String.new('foo', encoding: Encoding::US_ASCII).encoding # => #<Encoding:US-A
String.new('foo', encoding: 'US-ASCII').encoding         # => #<Encoding:US-A
String.new('foo', encoding: 'ASCII').encoding            # => #<Encoding:US-A
```

The given encoding need not be valid for the string's content, and that validity is not checked:

```
s = String.new('こんにちは', encoding: 'ascii')
s.valid_encoding? # => false
```

But the given `encoding` itself is checked:

```
String.new('foo', encoding: 'bar') # Raises ArgumentError.
```

With optional keyword argument `capacity`, returns a copy of `string` (or an empty string, if `string` is not given); the given `capacity` is advisory only, and may or may not set the size of the internal buffer, which may in turn affect performance:

```
String.new(capacity: 1)
String.new('foo', capacity: 4096)
```

The `string`, `encoding`, and `capacity` arguments may all be used together:

```
String.new('hello', encoding: 'UTF-8', capacity: 25)
```

try_convert(object) → object, new_string, or nil

If `object` is a String object, returns `object`.

Otherwise if `object` responds to `:to_str`, calls `object.to_str` and returns the result.

Returns `nil` if `object` does not respond to `:to_str`.

Raises an exception unless `object.to_str` returns a String object.

Public Instance Methods

`string % object → new_string`

Returns the result of formatting `object` into the format specification `self` (see `Kernel#sprintf` for formatting details):

```
"%05d" % 123 # => "00123"
```

If `self` contains multiple substitutions, `object` must be an `Array` or `Hash` containing the values to be substituted:

```
"%-5s: %016x" % [ "ID", self.object_id ] # => "ID   : 00002b054ec93168"
"foo = %{foo}" % {foo: 'bar'} # => "foo = bar"
"foo = %{foo}, baz = %{baz}" % {foo: 'bar', baz: 'bat'} # => "foo = bar, baz
```

`string * integer → new_string`

Returns a new String containing `integer` copies of `self`:

```
"Ho! " * 3 # => "Ho! Ho! Ho! "
"Ho! " * 0 # => ""
```

`string + other_string → new_string`

Returns a new String containing `other_string` concatenated to `self`:

```
"Hello from " + self.to_s # => "Hello from main"
```

`+string → new_string or self`

Returns `self` if `self` is not frozen.

Otherwise returns `self.dup`, which is not frozen.

-string → frozen_string

Returns a frozen, possibly pre-existing copy of the string.

The returned String will be deduplicated as long as it does not have any instance variables set on it and is not a `String` subclass.

Note that `-string` variant is more convenient for defining constants:

```
FILENAME = -'config/database.yml'
```

while `dedup` is better suitable for using the method in chains of calculations:

```
@url_list.concat(urls.map(&:dedup))
```

Also aliased as: dedup

string << object → string

Concatenates `object` to `self` and returns `self`:

```
s = 'foo'
s << 'bar' # => "foobar"
s          # => "foobar"
```

If `object` is an `Integer`, the value is considered a codepoint and converted to a character before concatenation:

```
s = 'foo'
s << 33 # => "foo!"
```

Related: `String#concat`, which takes multiple arguments.

string <=> other_string → -1, 0, 1, or nil

Compares `self` and `other_string`, returning:

- -1 if `other_string` is larger.
- 0 if the two are equal.
- 1 if `other_string` is smaller.
- `nil` if the two are incomparable.

Examples:

```
'foo' <=> 'foo' # => 0
'foo' <=> 'food' # => -1
'food' <=> 'foo' # => 1
'FOO' <=> 'foo' # => -1
'foo' <=> 'FOO' # => 1
'foo' <=> 1 # => nil
```

`string == object → true or false`

Returns `true` if `object` has the same length and content; as `self`; `false` otherwise:

```
s = 'foo'
s == 'foo' # => true
s == 'food' # => false
s == 'FOO' # => false
```

Returns `false` if the two strings' encodings are not compatible:

```
"\u{e4 f6 fc}".encode("ISO-8859-1") == ("\u{c4 d6 dc}") # => false
```

If `object` is not an instance of String but responds to `to_str`, then the two strings are compared using `object.==`.

Also aliased as: ===

`string === object → true or false`

Returns `true` if `object` has the same length and content; as `self`; `false` otherwise:

```
s = 'foo'
s == 'foo' # => true
s == 'food' # => false
s == 'FOO' # => false
```

Returns `false` if the two strings' encodings are not compatible:

```
"\u{e4 f6 fc}".encode("ISO-8859-1") == ("\u{c4 d6 dc}") # => false
```

If `object` is not an instance of String but responds to `to_str`, then the two strings are compared using `object.==`.

Alias for: ==

string =~ regexp → integer or nil
string =~ object → integer or nil

Returns the Integer index of the first substring that matches the given `regexp`, or `nil` if no match found:

```
'foo' =~ /f/ # => 0
'foo' =~ /o/ # => 1
'foo' =~ /x/ # => nil
```

Note: also updates Global Variables at Regexp.

If the given `object` is not a Regexp, returns the value returned by `object =~ self`.

Note that `string =~ regexp` is different from `regexp =~ string` (see Regexp#=~):

```
number= nil
"no. 9" =~ /(?<number>\d+)/
number # => nil (not assigned)
/(?<number>\d+)/ =~ "no. 9"
number #=> "9"
```

string[index] → new_string or nil
string[start, length] → new_string or nil
string[range] → new_string or nil
string[regexp, capture = 0] → new_string or nil
string[substring] → new_string or nil

Returns the substring of `self` specified by the arguments. See examples at String Slices.

Also aliased as: slice

string[index] = new_string
string[start, length] = new_string
string[range] = new_string
string[regexp, capture = 0] = new_string
string[substring] = new_string

Replaces all, some, or none of the contents of `self`; returns `new_string`. See String Slices.

A few examples:

```
s = 'foo'
s[2] = 'rtune'      # => "rtune"
s                   # => "fortune"
```

```
s[1, 5] = 'init'   # => "init"
s                  # => "finite"
s[3..4] = 'al'     # => "al"
s                  # => "finale"
s[/e$/] = 'ly'     # => "ly"
s                  # => "finally"
s['lly'] = 'ncial' # => "ncial"
s                  # => "financial"
```

ascii_only? → true or false

Returns `true` if `self` contains only ASCII characters, `false` otherwise:

```
'abc'.ascii_only?          # => true
"abc\u{6666}".ascii_only? # => false
```

b → string

Returns a copy of `self` that has ASCII-8BIT encoding; the underlying bytes are not modified:

```
s = "\x99"
s.encoding   # => #<Encoding:UTF-8>
t = s.b      # => "\x99"
t.encoding   # => #<Encoding:ASCII-8BIT>

s = "\u4095" # => "䂕"
s.encoding   # => #<Encoding:UTF-8>
s.bytes      # => [228, 130, 149]
t = s.b      # => "\xE4\x82\x95"
t.encoding   # => #<Encoding:ASCII-8BIT>
t.bytes      # => [228, 130, 149]
```

byteindex(substring, offset = 0) → integer or nil
byteindex(regexp, offset = 0) → integer or nil

Returns the Integer byte-based index of the first occurrence of the given `substring`, or `nil` if none found:

```
'foo'.byteindex('f') # => 0
'foo'.byteindex('o') # => 1
'foo'.byteindex('oo') # => 1
'foo'.byteindex('ooo') # => nil
```

Returns the Integer byte-based index of the first match for the given Regexp `regexp`, or `nil` if none found:

```
'foo'.byteindex(/f/) # => 0
'foo'.byteindex(/o/) # => 1
'foo'.byteindex(/oo/) # => 1
'foo'.byteindex(/ooo/) # => nil
```

Integer argument `offset`, if given, specifies the byte-based position in the string to begin the search:

```
'foo'.byteindex('o', 1) # => 1
'foo'.byteindex('o', 2) # => 2
'foo'.byteindex('o', 3) # => nil
```

If `offset` is negative, counts backward from the end of `self`:

```
'foo'.byteindex('o', -1) # => 2
'foo'.byteindex('o', -2) # => 1
'foo'.byteindex('o', -3) # => 1
'foo'.byteindex('o', -4) # => nil
```

If `offset` does not land on character (codepoint) boundary, `IndexError` is raised.

Related: String#index, String#byterindex.

byterindex(substring, offset = self.bytesize) → integer or nil
byterindex(regexp, offset = self.bytesize) → integer or nil

Returns the Integer byte-based index of the *last* occurrence of the given `substring`, or `nil` if none found:

```
'foo'.byterindex('f') # => 0
'foo'.byterindex('o') # => 2
'foo'.byterindex('oo') # => 1
'foo'.byterindex('ooo') # => nil
```

Returns the Integer byte-based index of the *last* match for the given Regexp `regexp`, or `nil` if none found:

```
'foo'.byterindex(/f/) # => 0
'foo'.byterindex(/o/) # => 2
'foo'.byterindex(/oo/) # => 1
'foo'.byterindex(/ooo/) # => nil
```

The *last* match means starting at the possible last position, not the last of longest matches.

```
'foo'.byterindex(/o+/) # => 2
```

```
$~ #=> #<MatchData "o">
```

To get the last longest match, needs to combine with negative lookbehind.

```
'foo'.byterindex(/(?<!o)o+/) # => 1
$~ #=> #<MatchData "oo">
```

Or `String#byteindex` with negative lookforward.

```
'foo'.byteindex(/o+(?!.*o)/) # => 1
$~ #=> #<MatchData "oo">
```

`Integer` argument `offset`, if given and non-negative, specifies the maximum starting byte-based position in the

```
string to _end_ the search:

 'foo'.byterindex('o', 0) # => nil
 'foo'.byterindex('o', 1) # => 1
 'foo'.byterindex('o', 2) # => 2
 'foo'.byterindex('o', 3) # => 2
```

If `offset` is a negative `Integer`, the maximum starting position in the string to *end* the search is the sum of the string's length and `offset`:

```
'foo'.byterindex('o', -1) # => 2
'foo'.byterindex('o', -2) # => 1
'foo'.byterindex('o', -3) # => nil
'foo'.byterindex('o', -4) # => nil
```

If `offset` does not land on character (codepoint) boundary, `IndexError` is raised.

Related: `String#byteindex`.

bytes → array_of_bytes

Returns an array of the bytes in `self`:

```
'hello'.bytes # => [104, 101, 108, 108, 111]
'тест'.bytes  # => [209, 130, 208, 181, 209, 129, 209, 130]
'こんにちは'.bytes
# => [227, 129, 147, 227, 130, 147, 227, 129, 171, 227, 129, 161, 227, 129, 1
```

bytesize → integer

Returns the count of bytes (not characters) in `self`:

```
'foo'.bytesize        # => 3
'тест'.bytesize       # => 8
'こんにちは'.bytesize   # => 15
```

Contrast with [String#length]():

```
'foo'.length       # => 3
'тест'.length      # => 4
'こんにちは'.length  # => 5
```

byteslice(index, length = 1) → string or nil
byteslice(range) → string or nil

Returns a substring of `self`, or `nil` if the substring cannot be constructed.

With integer arguments `index` and `length` given, returns the substring beginning at the given `index` of the given `length` (if possible), or `nil` if `length` is negative or `index` falls outside of `self`:

```
s = '0123456789' # => "0123456789"
s.byteslice(2)   # => "2"
s.byteslice(200) # => nil
s.byteslice(4, 3)  # => "456"
s.byteslice(4, 30) # => "456789"
s.byteslice(4, -1) # => nil
s.byteslice(40, 2) # => nil
```

In either case above, counts backwards from the end of `self` if `index` is negative:

```
s = '0123456789'   # => "0123456789"
s.byteslice(-4)    # => "6"
s.byteslice(-4, 3) # => "678"
```

With [Range]() argument `range` given, returns `byteslice(range.begin, range.size)`:

```
s = '0123456789'    # => "0123456789"
s.byteslice(4..6)   # => "456"
s.byteslice(-6..-4) # => "456"
s.byteslice(5..2)   # => "" # range.size is zero.
s.byteslice(40..42) # => nil
```

In all cases, a returned string has the same encoding as `self`:

```
s.encoding              # => #<Encoding:UTF-8>
s.byteslice(4).encoding # => #<Encoding:UTF-8>
```

bytesplice(index, length, str) → string
bytesplice(index, length, str, str_index, str_length) → string
bytesplice(range, str) → string
bytesplice(range, str, str_range) → string

Replaces some or all of the content of `self` with `str`, and returns `self`. The portion of the string affected is determined using the same criteria as `String#byteslice`, except that `length` cannot be omitted. If the replacement string is not the same length as the text it is replacing, the string will be adjusted accordingly.

If `str_index` and `str_length`, or `str_range` are given, the content of `self` is replaced by str.byteslice(str_index, str_length) or str.byteslice(str_range); however the substring of `str` is not allocated as a new string.

The form that take an `Integer` will raise an `IndexError` if the value is out of range; the `Range` form will raise a `RangeError`. If the beginning or ending offset does not land on character (codepoint) boundary, an `IndexError` will be raised.

capitalize(*options) → string

Returns a string containing the characters in `self`; the first character is upcased; the remaining characters are downcased:

```
s = 'hello World!' # => "hello World!"
s.capitalize       # => "Hello world!"
```

The casing may be affected by the given `options`; see Case Mapping.

Related: `String#capitalize!`.

capitalize!(*options) → self or nil

Upcases the first character in `self`; downcases the remaining characters; returns `self` if any changes were made, `nil` otherwise:

```
s = 'hello World!' # => "hello World!"
s.capitalize!      # => "Hello world!"
s                  # => "Hello world!"
s.capitalize!      # => nil
```

The casing may be affected by the given `options`; see Case Mapping.

Related: `String#capitalize`.

casecmp(other_string) → -1, 0, 1, or nil

Compares `self.downcase` and `other_string.downcase`; returns:

- -1 if `other_string.downcase` is larger.
- 0 if the two are equal.
- 1 if `other_string.downcase` is smaller.
- `nil` if the two are incomparable.

Examples:

```
'foo'.casecmp('foo') # => 0
'foo'.casecmp('food') # => -1
'food'.casecmp('foo') # => 1
'FOO'.casecmp('foo') # => 0
'foo'.casecmp('FOO') # => 0
'foo'.casecmp(1) # => nil
```

See Case Mapping.

Related: `String#casecmp?`.

`casecmp?(other_string) → true, false, or nil`

Returns `true` if `self` and `other_string` are equal after Unicode case folding, otherwise `false`:

```
'foo'.casecmp?('foo') # => true
'foo'.casecmp?('food') # => false
'food'.casecmp?('foo') # => false
'FOO'.casecmp?('foo') # => true
'foo'.casecmp?('FOO') # => true
```

Returns `nil` if the two values are incomparable:

```
'foo'.casecmp?(1) # => nil
```

See Case Mapping.

Related: `String#casecmp`.

`center(size, pad_string = ' ') → new_string`

Returns a centered copy of `self`.

If integer argument `size` is greater than the size (in characters) of `self`, returns a new string of length `size` that is a copy of `self`, centered and padded on both ends with `pad_string`:

```
'hello'.center(10)       # => "  hello   "
'  hello'.center(10)     # => "   hello  "
'hello'.center(10, 'ab') # => "abhelloaba"
```

```
'тест'.center(10)        # => "   тест   "
'こんにちは'.center(10)    # => "  こんにちは   "
```

If `size` is not greater than the size of `self`, returns a copy of `self`:

```
'hello'.center(5)  # => "hello"
'hello'.center(1)  # => "hello"
```

Related: [String#ljust](), [String#rjust]().

chars → array_of_characters

Returns an array of the characters in `self`:

```
'hello'.chars     # => ["h", "e", "l", "l", "o"]
'тест'.chars      # => ["т", "е", "с", "т"]
'こんにちは'.chars # => ["こ", "ん", "に", "ち", "は"]
```

chomp(line_sep = $/) → new_string

Returns a new string copied from `self`, with trailing characters possibly removed:

When `line_sep` is `"\n"`, removes the last one or two characters if they are `"\r"`, `"\n"`, or `"\r\n"` (but not `"\n\r"`):

```
$/                    # => "\n"
"abc\r".chomp         # => "abc"
"abc\n".chomp         # => "abc"
"abc\r\n".chomp       # => "abc"
"abc\n\r".chomp       # => "abc\n"
"тест\r\n".chomp      # => "тест"
"こんにちは\r\n".chomp  # => "こんにちは"
```

When `line_sep` is `''` (an empty string), removes multiple trailing occurrences of `"\n"` or `"\r\n"` (but not `"\r"` or `"\n\r"`):

```
"abc\n\n\n".chomp('')           # => "abc"
"abc\r\n\r\n\r\n".chomp('')     # => "abc"
"abc\n\n\r\n\r\n\n\n".chomp('') # => "abc"
"abc\n\r\n\r\n\r".chomp('')     # => "abc\n\r\n\r\n\r"
"abc\r\r\r".chomp('')           # => "abc\r\r\r"
```

When `line_sep` is neither `"\n"` nor `''`, removes a single trailing line separator if there is one:

```
'abcd'.chomp('d')  # => "abc"
'abcdd'.chomp('d') # => "abcd"
```

chomp!(line_sep = $/) → self or nil

Like String#chomp, but modifies self in place; returns nil if no modification made, self otherwise.

chop → new_string

Returns a new string copied from self, with trailing characters possibly removed.

Removes "\r\n" if those are the last two characters.

```
"abc\r\n".chop      # => "abc"
"тест\r\n".chop     # => "тест"
"こんにちは\r\n".chop # => "こんにちは"
```

Otherwise removes the last character if it exists.

```
'abcd'.chop     # => "abc"
'тест'.chop     # => "тес"
'こんにちは'.chop # => "こんにち"
''.chop         # => ""
```

If you only need to remove the newline separator at the end of the string, String#chomp is a better alternative.

chop! → self or nil

Like String#chop, but modifies self in place; returns nil if self is empty, self otherwise.

Related: String#chomp!.

chr → string

Returns a string containing the first character of self:

```
s = 'foo' # => "foo"
s.chr     # => "f"
```

clear → self

Removes the contents of self:

```
s = 'foo' # => "foo"
s.clear   # => ""
```

codepoints → array_of_integers

Returns an array of the codepoints in `self`; each codepoint is the integer value for a character:

```
'hello'.codepoints     # => [104, 101, 108, 108, 111]
'тест'.codepoints      # => [1090, 1077, 1089, 1090]
'こんにちは'.codepoints # => [12371, 12435, 12395, 12385, 12399]
```

concat(*objects) → string

Concatenates each object in `objects` to `self` and returns `self`:

```
s = 'foo'
s.concat('bar', 'baz') # => "foobarbaz"
s                      # => "foobarbaz"
```

For each given object `object` that is an `Integer`, the value is considered a codepoint and converted to a character before concatenation:

```
s = 'foo'
s.concat(32, 'bar', 32, 'baz') # => "foo bar baz"
```

Related: `String#<<`, which takes a single argument.

count(*selectors) → integer

Returns the total number of characters in `self` that are specified by the given `selectors` (see Multiple Character Selectors):

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

"hello^world".count "\\^aeiou" #=> 4
"hello-world".count "a\\-eo"   #=> 4

c = "hello world\\r\\n"
c.count "\\"                   #=> 2
c.count "\\A"                  #=> 0
c.count "X-\\w"                #=> 3
```

`crypt(salt_str) → new_string`

Returns the string generated by calling `crypt(3)` standard library function with `str` and `salt_str`, in this order, as its arguments. Please do not use this method any longer. It is legacy; provided only for backward compatibility with ruby scripts in earlier days. It is bad to use in contemporary programs for several reasons:

- Behaviour of C's `crypt(3)` depends on the OS it is run. The generated string lacks data portability.
- On some OSes such as Mac OS, `crypt(3)` never fails (i.e. silently ends up in unexpected results).
- On some OSes such as Mac OS, `crypt(3)` is not thread safe.
- So-called "traditional" usage of `crypt(3)` is very very very weak. According to its manpage, Linux's traditional `crypt(3)` output has only 2**56 variations; too easy to brute force today. And this is the default behaviour.
- In order to make things robust some OSes implement so-called "modular" usage. To go through, you have to do a complex build-up of the `salt_str` parameter, by hand. Failure in generation of a proper salt string tends not to yield any errors; typos in parameters are normally not detectable.
    - For instance, in the following example, the second invocation of `String#crypt` is wrong; it has a typo in "round=" (lacks "s"). However the call does not fail and something unexpected is generated.

```
"foo".crypt("$5$rounds=1000$salt$") # OK, proper usage
"foo".crypt("$5$round=1000$salt$")  # Typo not detected
```

- Even in the "modular" mode, some hash functions are considered archaic and no longer recommended at all; for instance module `$1$` is officially abandoned by its author: see phk.freebsd.dk/sagas/md5crypt_eol/ . For another instance module `$3$` is considered completely broken: see the manpage of FreeBSD.
- On some OS such as Mac OS, there is no modular mode. Yet, as written above, `crypt(3)` on Mac OS never fails. This means even if you build up a proper salt string it generates a traditional DES hash anyways, and there is no way for you to be aware of.

```
"foo".crypt("$5$rounds=1000$salt$") # => "$5fNPQMxC5j6."
```

If for some reason you cannot migrate to other secure contemporary password hashing algorithms, install the string-crypt gem and `require 'string/crypt'` to continue using it.

-string → frozen_string
dedup → frozen_string

Returns a frozen, possibly pre-existing copy of the string.

The returned String will be deduplicated as long as it does not have any instance variables set on it and is not a `String` subclass.

Note that `-string` variant is more convenient for defining constants:

```
FILENAME = -'config/database.yml'
```

while `dedup` is better suitable for using the method in chains of calculations:

```
@url_list.concat(urls.map(&:dedup))
```

Alias for: -@

delete(*selectors) → new_string

Returns a copy of `self` with characters specified by `selectors` removed (see Multiple Character Selectors):

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

delete!(*selectors) → self or nil

Like `String#delete`, but modifies `self` in place. Returns `self` if any changes were made, `nil` otherwise.

delete_prefix(prefix) → new_string

Returns a copy of `self` with leading substring `prefix` removed:

```
'hello'.delete_prefix('hel')       # => "lo"
'hello'.delete_prefix('llo')       # => "hello"
'тест'.delete_prefix('те')         # => "ст"
'こんにちは'.delete_prefix('こん')  # => "にちは"
```

Related: `String#delete_prefix!`, `String#delete_suffix`.

delete_prefix!(prefix) → self or nil

Like String#delete_prefix, except that self is modified in place. Returns self if the prefix is removed, nil otherwise.

delete_suffix(suffix) → new_string

Returns a copy of self with trailing substring suffix removed:

```
'hello'.delete_suffix('llo')      # => "he"
'hello'.delete_suffix('hel')      # => "hello"
'тест'.delete_suffix('ст')        # => "те"
'こんにちは'.delete_suffix('ちは')  # => "こんに"
```

Related: String#delete_suffix!, String#delete_prefix.

delete_suffix!(suffix) → self or nil

Like String#delete_suffix, except that self is modified in place. Returns self if the suffix is removed, nil otherwise.

downcase(*options) → string

Returns a string containing the downcased characters in self:

```
s = 'Hello World!' # => "Hello World!"
s.downcase         # => "hello world!"
```

The casing may be affected by the given options; see Case Mapping.

Related: String#downcase!, String#upcase, String#upcase!.

downcase!(*options) → self or nil

Downcases the characters in self; returns self if any changes were made, nil otherwise:

```
s = 'Hello World!' # => "Hello World!"
s.downcase!        # => "hello world!"
s                  # => "hello world!"
s.downcase!        # => nil
```

The casing may be affected by the given options; see Case Mapping.

Related: String#downcase, String#upcase, String#upcase!.

dump → string

Returns a printable version of `self`, enclosed in double-quotes, with special characters escaped, and with non-printing characters replaced by hexadecimal notation:

```
"hello \n ''".dump    # => "\"hello \\n ''\""
"\f\x00\xff\\\"".dump # => "\"\\f\\x00\\xFF\\\\\\\"\""
```

Related: `String#undump` (inverse of `String#dump`).

each_byte {|byte| ... } → self
each_byte → enumerator

Calls the given block with each successive byte from `self`; returns `self`:

```
'hello'.each_byte {|byte| print byte, ' ' }
print "\n"
'тест'.each_byte {|byte| print byte, ' ' }
print "\n"
'こんにちは'.each_byte {|byte| print byte, ' ' }
print "\n"
```

Output:

```
104 101 108 108 111
209 130 208 181 209 129 209 130
227 129 147 227 130 147 227 129 171 227 129 161 227 129 175
```

Returns an enumerator if no block is given.

each_char {|c| ... } → self
each_char → enumerator

Calls the given block with each successive character from `self`; returns `self`:

```
'hello'.each_char {|char| print char, ' ' }
print "\n"
'тест'.each_char {|char| print char, ' ' }
print "\n"
'こんにちは'.each_char {|char| print char, ' ' }
print "\n"
```

Output:

```
h e l l o
т е с т
こ ん に ち は
```

Returns an enumerator if no block is given.

each_codepoint {|integer| ... } → self
each_codepoint → enumerator

Calls the given block with each successive codepoint from `self`; each codepoint is the integer value for a character; returns `self`:

```
'hello'.each_codepoint {|codepoint| print codepoint, ' ' }
print "\n"
'тест'.each_codepoint {|codepoint| print codepoint, ' ' }
print "\n"
'こんにちは'.each_codepoint {|codepoint| print codepoint, ' ' }
print "\n"
```

Output:

```
104 101 108 108 111
1090 1077 1089 1090
12371 12435 12395 12385 12399
```

Returns an enumerator if no block is given.

each_grapheme_cluster {|gc| ... } → self
each_grapheme_cluster → enumerator

Calls the given block with each successive grapheme cluster from `self` (see Unicode Grapheme Cluster Boundaries); returns `self`:

```
s = "\u0061\u0308-pqr-\u0062\u0308-xyz-\u0063\u0308" # => "ä-pqr-b̈-xyz-c̈"
s.each_grapheme_cluster {|gc| print gc, ' ' }
```

Output:

```
ä - p q r - b̈ - x y z - c̈
```

Returns an enumerator if no block is given.

each_line(line_sep = $/, chomp: false) {|substring| ... } → self
each_line(line_sep = $/, chomp: false) → enumerator

With a block given, forms the substrings (“lines”) that are the result of splitting `self` at each occurrence of the given line separator `line_sep`; passes each line to the block; returns `self`:

```
s = <<~EOT
This is the first line.
This is line two.

This is line four.
This is line five.
EOT

s.each_line {|line| p line }
```

Output:

```
"This is the first line.\n"
"This is line two.\n"
"\n"
"This is line four.\n"
"This is line five.\n"
```

With a different `line_sep`:

```
s.each_line(' is ') {|line| p line }
```

Output:

```
"This is "
"the first line.\nThis is "
"line two.\n\nThis is "
"line four.\nThis is "
"line five.\n"
```

With `chomp` as `true`, removes the trailing `line_sep` from each line:

```
s.each_line(chomp: true) {|line| p line }
```

Output:

```
"This is the first line."
"This is line two."
""
"This is line four."
"This is line five."
```

With an empty string as `line_sep`, forms and passes “paragraphs” by splitting at each occurrence of two or more newlines:

```
s.each_line('') {|line| p line }
```

Output:

```
"This is the first line.\nThis is line two.\n\n"
"This is line four.\nThis is line five.\n"
```

With no block given, returns an enumerator.

`empty? → true or false`

Returns `true` if the length of `self` is zero, `false` otherwise:

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

`encode(dst_encoding = Encoding.default_internal, **enc_opts) → string`
`encode(dst_encoding, src_encoding, **enc_opts) → string`

Returns a copy of `self` transcoded as determined by `dst_encoding`. By default, raises an exception if `self` contains an invalid byte or a character not defined in `dst_encoding`; that behavior may be modified by encoding options; see below.

With no arguments:

- Uses the same encoding if `Encoding.default_internal` is `nil` (the default):

```
Encoding.default_internal # => nil
s = "Ruby\x99".force_encoding('Windows-1252')
s.encoding                # => #<Encoding:Windows-1252>
s.bytes                   # => [82, 117, 98, 121, 153]
t = s.encode              # => "Ruby\x99"
t.encoding                # => #<Encoding:Windows-1252>
t.bytes                   # => [82, 117, 98, 121, 226, 132, 162]
```

- Otherwise, uses the encoding `Encoding.default_internal`:

```
Encoding.default_internal = 'UTF-8'
t = s.encode              # => "Ruby™"
t.encoding                # => #<Encoding:UTF-8>
```

With only argument `dst_encoding` given, uses that encoding:

```
s = "Ruby\x99".force_encoding('Windows-1252')
s.encoding            # => #<Encoding:Windows-1252>
t = s.encode('UTF-8') # => "Ruby™"
t.encoding            # => #<Encoding:UTF-8>
```

With arguments `dst_encoding` and `src_encoding` given, interprets `self` using `src_encoding`, encodes the new string using `dst_encoding`:

```
s = "Ruby\x99"
t = s.encode('UTF-8', 'Windows-1252') # => "Ruby™"
t.encoding                            # => #<Encoding:UTF-8>
```

Optional keyword arguments `enc_opts` specify encoding options; see Encoding Options.

Please note that, unless `invalid: :replace` option is given, conversion from an encoding `enc` to the same encoding `enc` (independent of whether `enc` is given explicitly or implicitly) is a no-op, i.e. the string is simply copied without any changes, and no exceptions are raised, even if there are invalid bytes.

encode!(dst_encoding = Encoding.default_internal, **enc_opts) → self
encode!(dst_encoding, src_encoding, **enc_opts) → self

Like `encode`, but applies encoding changes to `self`; returns `self`.

encoding → encoding

Returns the `Encoding` object that represents the encoding of obj.

end_with?(*strings) → true or false

Returns whether `self` ends with any of the given `strings`.

Returns `true` if any given string matches the end, `false` otherwise:

```
'hello'.end_with?('ello')               #=> true
'hello'.end_with?('heaven', 'ello')     #=> true
'hello'.end_with?('heaven', 'paradise') #=> false
'тест'.end_with?('т')                   # => true
'こんにちは'.end_with?('は')             # => true
```

Related: `String#start_with?`.

eql?(object) → true or false

Returns `true` if `object` has the same length and content; as `self`; `false` otherwise:

```
s = 'foo'
s.eql?('foo') # => true
```

```
s.eql?('food') # => false
s.eql?('FOO') # => false
```

Returns `false` if the two strings' encodings are not compatible:

```
"\u{e4 f6 fc}".encode("ISO-8859-1").eql?("\u{c4 d6 dc}") # => false
```

`force_encoding(encoding) → self`

Changes the encoding of `self` to `encoding`, which may be a string encoding name or an `Encoding` object; returns self:

```
s = 'ŁAŁ'
s.bytes                    # => [197, 130, 97, 197, 130]
s.encoding                 # => #<Encoding:UTF-8>
s.force_encoding('ascii') # => "\xC5\x82a\xC5\x82"
s.encoding                 # => #<Encoding:US-ASCII>
```

Does not change the underlying bytes:

```
s.bytes                    # => [197, 130, 97, 197, 130]
```

Makes the change even if the given `encoding` is invalid for `self` (as is the change above):

```
s.valid_encoding?                  # => false
s.force_encoding(Encoding::UTF_8) # => "ŁAŁ"
s.valid_encoding?                  # => true
```

`getbyte(index) → integer or nil`

Returns the byte at zero-based `index` as an integer, or `nil` if `index` is out of range:

```
s = 'abcde'   # => "abcde"
s.getbyte(0)  # => 97
s.getbyte(-1) # => 101
s.getbyte(5)  # => nil
```

Related: `String#setbyte`.

`grapheme_clusters → array_of_grapheme_clusters`

Returns an array of the grapheme clusters in `self` (see Unicode Grapheme Cluster Boundaries):

```
s = "\u0061\u0308-pqr-\u0062\u0308-xyz-\u0063\u0308" # => "ä-pqr-b̈-xyz-c̈"
s.grapheme_clusters
# => ["ä", "-", "p", "q", "r", "-", "b̈", "-", "x", "y", "z", "-", "c̈"]
```

gsub(pattern, replacement) → new_string
gsub(pattern) {|match| ... } → new_string
gsub(pattern) → enumerator

Returns a copy of `self` with all occurrences of the given `pattern` replaced.

See Substitution Methods.

Returns an `Enumerator` if no `replacement` and no block given.

Related: `String#sub`, `String#sub!`, `String#gsub!`.

gsub!(pattern, replacement) → self or nil
gsub!(pattern) {|match| ... } → self or nil
gsub!(pattern) → an_enumerator

Performs the specified substring replacement(s) on `self`; returns `self` if any replacement occurred, `nil` otherwise.

See Substitution Methods.

Returns an `Enumerator` if no `replacement` and no block given.

Related: `String#sub`, `String#gsub`, `String#sub!`.

hash → integer

Returns the integer hash value for `self`. The value is based on the length, content and encoding of `self`.

Related: `Object#hash`.

hex → integer

Interprets the leading substring of `self` as a string of hexadecimal digits (with an optional sign and an optional `0x`) and returns the corresponding number; returns zero if there is no such leading substring:

```
'0x0a'.hex        # => 10
'-1234'.hex       # => -4660
'0'.hex           # => 0
'non-numeric'.hex # => 0
```

Related: `String#oct`.

include? other_string → true or false

Returns `true` if `self` contains `other_string`, `false` otherwise:

```
s = 'foo'
s.include?('f')    # => true
s.include?('fo')   # => true
s.include?('food') # => false
```

index(substring, offset = 0) → integer or nil
index(regexp, offset = 0) → integer or nil

Returns the integer index of the first match for the given argument, or `nil` if none found; the search of `self` is forward, and begins at position `offset` (in characters).

With string argument `substring`, returns the index of the first matching substring in `self`:

```
'foo'.index('f')          # => 0
'foo'.index('o')          # => 1
'foo'.index('oo')         # => 1
'foo'.index('ooo')        # => nil
'тест'.index('с')         # => 2
'こんにちは'.index('ち')   # => 3
```

With Regexp argument `regexp`, returns the index of the first match in `self`:

```
'foo'.index(/o./) # => 1
'foo'.index(/.o/) # => 0
```

With positive integer `offset`, begins the search at position `offset`:

```
'foo'.index('o', 1)         # => 1
'foo'.index('o', 2)         # => 2
'foo'.index('o', 3)         # => nil
'тест'.index('с', 1)        # => 2
'こんにちは'.index('ち', 2)  # => 3
```

With negative integer `offset`, selects the search position by counting backward from the end of `self`:

```
'foo'.index('o', -1)  # => 2
'foo'.index('o', -2)  # => 1
'foo'.index('o', -3)  # => 1
'foo'.index('o', -4)  # => nil
'foo'.index(/o./, -2) # => 1
'foo'.index(/.o/, -2) # => 1
```

Related: String#rindex.

initialize_copy(other_string) -> self

Replaces the contents of `self` with the contents of `other_string`:

```
s = 'foo'         # => "foo"
s.replace('bar') # => "bar"
```

Also aliased as: replace

insert(index, other_string) → self

Inserts the given `other_string` into `self`; returns `self`.

If the `Integer` `index` is positive, inserts `other_string` at offset `index`:

```
'foo'.insert(1, 'bar') # => "fbaroo"
```

If the `Integer` `index` is negative, counts backward from the end of `self` and inserts `other_string` at offset `index+1` (that is, *after* `self[index]`):

```
'foo'.insert(-2, 'bar') # => "fobaro"
```

inspect → string

Returns a printable version of `self`, enclosed in double-quotes, and with special characters escaped:

```
s = "foo\tbar\tbaz\n"
s.inspect
# => "\"foo\\tbar\\tbaz\\n\""
```

intern → symbol

Returns the `Symbol` corresponding to *str*, creating the symbol if it did not previously exist. See `Symbol#id2name`.

```
"Koala".intern         #=> :Koala
s = 'cat'.to_sym       #=> :cat
s == :cat              #=> true
s = '@cat'.to_sym      #=> :@cat
s == :@cat             #=> true
```

This can also be used to create symbols that cannot be represented using the `:xxx` notation.

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

Also aliased as: to_sym

length → integer

Returns the count of characters (not bytes) in `self`:

```
'foo'.length        # => 3
'тест'.length       # => 4
'こんにちは'.length   # => 5
```

Contrast with `String#bytesize`:

```
'foo'.bytesize        # => 3
'тест'.bytesize       # => 8
'こんにちは'.bytesize   # => 15
```

Also aliased as: size

lines(Line_sep = $/, chomp: false) → array_of_strings

Forms substrings ("lines") of `self` according to the given arguments (see `String#each_line` for details); returns the lines in an array.

ljust(size, pad_string = ' ') → new_string

Returns a left-justified copy of `self`.

If integer argument `size` is greater than the size (in characters) of `self`, returns a new string of length `size` that is a copy of `self`, left justified and padded on the right with `pad_string`:

```
'hello'.ljust(10)       # => "hello     "
'  hello'.ljust(10)     # => "  hello   "
'hello'.ljust(10, 'ab') # => "helloababa"
'тест'.ljust(10)        # => "тест      "
'こんにちは'.ljust(10)    # => "こんにちは     "
```

If `size` is not greater than the size of `self`, returns a copy of `self`:

```
'hello'.ljust(5)  # => "hello"
'hello'.ljust(1)  # => "hello"
```

Related: String#rjust, String#center.

lstrip → new_string

Returns a copy of self with leading whitespace removed; see Whitespace in Strings:

```
whitespace = "\x00\t\n\v\f\r "
s = whitespace + 'abc' + whitespace
s        # => "\u0000\t\n\v\f\r abc\u0000\t\n\v\f\r "
s.lstrip # => "abc\u0000\t\n\v\f\r "
```

Related: String#rstrip, String#strip.

lstrip! → self or nil

Like String#lstrip, except that any modifications are made in self; returns self if any modification are made, nil otherwise.

Related: String#rstrip!, String#strip!.

match(pattern, offset = 0) → matchdata or nil
match(pattern, offset = 0) {|matchdata| ... } → object

Returns a MatchData object (or nil) based on self and the given pattern.

Note: also updates Global Variables at Regexp.

- Computes regexp by converting pattern (if not already a Regexp).

  ```
  regexp = Regexp.new(pattern)
  ```

- Computes matchdata, which will be either a MatchData object or nil (see Regexp#match):

  ```
  matchdata = <tt>regexp.match(self)
  ```

With no block given, returns the computed matchdata:

```
'foo'.match('f') # => #<MatchData "f">
'foo'.match('o') # => #<MatchData "o">
'foo'.match('x') # => nil
```

If Integer argument offset is given, the search begins at index offset:

```
'foo'.match('f', 1) # => nil
'foo'.match('o', 1) # => #<MatchData "o">
```

With a block given, calls the block with the computed `matchdata` and returns the block's return value:

```
'foo'.match(/o/) {|matchdata| matchdata } # => #<MatchData "o">
'foo'.match(/x/) {|matchdata| matchdata } # => nil
'foo'.match(/f/, 1) {|matchdata| matchdata } # => nil
```

match?(pattern, offset = 0) → true or false

Returns `true` or `false` based on whether a match is found for `self` and `pattern`.

Note: does not update Global Variables at `Regexp`.

Computes `regexp` by converting `pattern` (if not already a `Regexp`).

```
regexp = Regexp.new(pattern)
```

Returns `true` if `self+.match(regexp)` returns a `MatchData` object, `false` otherwise:

```
'foo'.match?(/o/) # => true
'foo'.match?('o') # => true
'foo'.match?(/x/) # => false
```

If `Integer` argument `offset` is given, the search begins at index `offset`:

```
'foo'.match?('f', 1) # => false
'foo'.match?('o', 1) # => true
```

next()

Returns the successor to `self`. The successor is calculated by incrementing characters.

The first character to be incremented is the rightmost alphanumeric: or, if no alphanumerics, the rightmost character:

```
'THX1138'.succ # => "THX1139"
'<<koala>>'.succ # => "<<koalb>>"
'***'.succ # => '**+'
```

The successor to a digit is another digit, "carrying" to the next-left character for a "rollover" from 9 to 0, and prepending another digit if necessary:

```
'00'.succ # => "01"
'09'.succ # => "10"
'99'.succ # => "100"
```

The successor to a letter is another letter of the same case, carrying to the next-left character for a rollover, and prepending another same-case letter if necessary:

```
'aa'.succ # => "ab"
'az'.succ # => "ba"
'zz'.succ # => "aaa"
'AA'.succ # => "AB"
'AZ'.succ # => "BA"
'ZZ'.succ # => "AAA"
```

The successor to a non-alphanumeric character is the next character in the underlying character set's collating sequence, carrying to the next-left character for a rollover, and prepending another character if necessary:

```
s = 0.chr * 3
s # => "\x00\x00\x00"
s.succ # => "\x00\x00\x01"
s = 255.chr * 3
s # => "\xFF\xFF\xFF"
s.succ # => "\x01\x00\x00\x00"
```

Carrying can occur between and among mixtures of alphanumeric characters:

```
s = 'zz99zz99'
s.succ # => "aaa00aa00"
s = '99zz99zz'
s.succ # => "100aa00aa"
```

The successor to an empty String is a new empty String:

```
''.succ # => ""
```

Alias for: succ

next!()

Equivalent to `String#succ`, but modifies `self` in place; returns `self`.

Alias for: succ!

oct → integer

Interprets the leading substring of `self` as a string of octal digits (with an optional sign) and returns the corresponding number; returns zero if there is no such leading

substring:

```
'123'.oct             # => 83
'-377'.oct            # => -255
'0377non-numeric'.oct # => 255
'non-numeric'.oct     # => 0
```

If `self` starts with `0`, radix indicators are honored; see `Kernel#Integer`.

Related: `String#hex`.

ord → integer

Returns the integer ordinal of the first character of `self`:

```
'h'.ord         # => 104
'hello'.ord     # => 104
'тест'.ord      # => 1090
'こんにちは'.ord  # => 12371
```

partition(string_or_regexp) → [head, match, tail]

Returns a 3-element array of substrings of `self`.

Matches a pattern against `self`, scanning from the beginning. The pattern is:

- `string_or_regexp` itself, if it is a `Regexp`.
- `Regexp.quote(string_or_regexp)`, if `string_or_regexp` is a string.

If the pattern is matched, returns pre-match, first-match, post-match:

```
'hello'.partition('l')       # => ["he", "l", "lo"]
'hello'.partition('ll')      # => ["he", "ll", "o"]
'hello'.partition('h')       # => ["", "h", "ello"]
'hello'.partition('o')       # => ["hell", "o", ""]
'hello'.partition(/l+/)      #=> ["he", "ll", "o"]
'hello'.partition('')        # => ["", "", "hello"]
'тест'.partition('т')        # => ["", "т", "ест"]
'こんにちは'.partition('に')  # => ["こん", "に", "ちは"]
```

If the pattern is not matched, returns a copy of `self` and two empty strings:

```
'hello'.partition('x') # => ["hello", "", ""]
```

Related: `String#rpartition`, `String#split`.

prepend(*other_strings) → string

Prepends each string in `other_strings` to `self` and returns `self`:

```
s = 'foo'
s.prepend('bar', 'baz') # => "barbazfoo"
s                       # => "barbazfoo"
```

Related: `String#concat`.

replace(other_string) → self

Replaces the contents of `self` with the contents of `other_string`:

```
s = 'foo'         # => "foo"
s.replace('bar') # => "bar"
```

Alias for: initialize_copy

reverse → string

Returns a new string with the characters from `self` in reverse order.

```
'stressed'.reverse # => "desserts"
```

reverse! → self

Returns `self` with its characters reversed:

```
s = 'stressed'
s.reverse! # => "desserts"
s          # => "desserts"
```

rindex(substring, offset = self.length) → integer or nil
rindex(regexp, offset = self.length) → integer or nil

Returns the `Integer` index of the *last* occurrence of the given `substring`, or `nil` if none found:

```
'foo'.rindex('f') # => 0
'foo'.rindex('o') # => 2
'foo'.rindex('oo') # => 1
'foo'.rindex('ooo') # => nil
```

Returns the Integer index of the *last* match for the given Regexp `regexp`, or `nil` if none found:

```
'foo'.rindex(/f/) # => 0
'foo'.rindex(/o/) # => 2
'foo'.rindex(/oo/) # => 1
'foo'.rindex(/ooo/) # => nil
```

The *last* match means starting at the possible last position, not the last of longest matches.

```
'foo'.rindex(/o+/) # => 2
$~ #=> #<MatchData "o">
```

To get the last longest match, needs to combine with negative lookbehind.

```
'foo'.rindex(/(?<!o)o+/) # => 1
$~ #=> #<MatchData "oo">
```

Or String#index with negative lookforward.

```
'foo'.index(/o+(?!.*o)/) # => 1
$~ #=> #<MatchData "oo">
```

Integer argument `offset`, if given and non-negative, specifies the maximum starting position in the

```
string to _end_ the search:

 'foo'.rindex('o', 0) # => nil
 'foo'.rindex('o', 1) # => 1
 'foo'.rindex('o', 2) # => 2
 'foo'.rindex('o', 3) # => 2
```

If `offset` is a negative Integer, the maximum starting position in the string to *end* the search is the sum of the string's length and `offset`:

```
'foo'.rindex('o', -1) # => 2
'foo'.rindex('o', -2) # => 1
'foo'.rindex('o', -3) # => nil
'foo'.rindex('o', -4) # => nil
```

Related: String#index.

rjust(size, pad_string = ' ') → new_string

Returns a right-justified copy of `self`.

If integer argument `size` is greater than the size (in characters) of `self`, returns a new string of length `size` that is a copy of `self`, right justified and padded on the left with `pad_string`:

```
'hello'.rjust(10)       # => "     hello"
'hello  '.rjust(10)     # => "   hello  "
'hello'.rjust(10, 'ab') # => "ababahello"
'тест'.rjust(10)        # => "      тест"
'こんにちは'.rjust(10)    # => "     こんにちは"
```

If `size` is not greater than the size of `self`, returns a copy of `self`:

```
'hello'.rjust(5, 'ab')  # => "hello"
'hello'.rjust(1, 'ab')  # => "hello"
```

Related: String#ljust, String#center.

rpartition(sep) → [head, match, tail]

Returns a 3-element array of substrings of `self`.

Matches a pattern against `self`, scanning backwards from the end. The pattern is:

- `string_or_regexp` itself, if it is a Regexp.
- `Regexp.quote(string_or_regexp)`, if `string_or_regexp` is a string.

If the pattern is matched, returns pre-match, last-match, post-match:

```
'hello'.rpartition('l')       # => ["hel", "l", "o"]
'hello'.rpartition('ll')      # => ["he", "ll", "o"]
'hello'.rpartition('h')       # => ["", "h", "ello"]
'hello'.rpartition('o')       # => ["hell", "o", ""]
'hello'.rpartition(/l+/)      # => ["hel", "l", "o"]
'hello'.rpartition('')        # => ["hello", "", ""]
'тест'.rpartition('т')        # => ["тес", "т", ""]
'こんにちは'.rpartition('に')  # => ["こん", "に", "ちは"]
```

If the pattern is not matched, returns two empty strings and a copy of `self`:

```
'hello'.rpartition('x') # => ["", "", "hello"]
```

Related: String#partition, String#split.

rstrip → new_string

Returns a copy of the receiver with trailing whitespace removed; see Whitespace in Strings:

```
whitespace = "\x00\t\n\v\f\r "
s = whitespace + 'abc' + whitespace
s        # => "\u0000\t\n\v\f\r abc\u0000\t\n\v\f\r "
s.rstrip # => "\u0000\t\n\v\f\r abc"
```

Related: String#lstrip, String#strip.

rstrip! → self or nil

Like String#rstrip, except that any modifications are made in `self`; returns `self` if any modification are made, `nil` otherwise.

Related: String#lstrip!, String#strip!.

scan(string_or_regexp) → array
scan(string_or_regexp) {|matches| ... } → self

Matches a pattern against `self`; the pattern is:

- `string_or_regexp` itself, if it is a Regexp.
- `Regexp.quote(string_or_regexp)`, if `string_or_regexp` is a string.

Iterates through `self`, generating a collection of matching results:

- If the pattern contains no groups, each result is the matched string, `$&`.
- If the pattern contains groups, each result is an array containing one entry per group.

With no block given, returns an array of the results:

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

With a block given, calls the block with each result; returns `self`:

```
s.scan(/\w+/) {|w| print "<<#{w}>> " }
print "\n"
s.scan(/(.)(.)/) {|x,y| print y, x }
print "\n"
```

Output:

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

scrub(replacement_string = default_replacement) → new_string
scrub{|bytes| ... } → new_string

Returns a copy of `self` with each invalid byte sequence replaced by the given `replacement_string`.

With no block given and no argument, replaces each invalid sequence with the default replacement string (`"\uFFFD"` for a Unicode encoding, `'?'` otherwise):

```
s = "foo\x81\x81bar"
s.scrub # => "foo\uFFFD\uFFFDbar"
```

With no block given and argument `replacement_string` given, replaces each invalid sequence with that string:

```
"foo\x81\x81bar".scrub('xyzzy') # => "fooxyzzyxyzzybar"
```

With a block given, replaces each invalid sequence with the value of the block:

```
"foo\x81\x81bar".scrub {|bytes| p bytes; 'XYZZY' }
# => "fooXYZZYXYZZYbar"
```

Output:

```
"\x81"
"\x81"
```

scrub! → self
scrub!(replacement_string = default_replacement) → self
scrub!{|bytes| ... } → self

Like `String#scrub`, except that any replacements are made in `self`.

setbyte(index, integer) → integer

Sets the byte at zero-based `index` to `integer`; returns `integer`:

```
s = 'abcde'       # => "abcde"
s.setbyte(0, 98) # => 98
s                 # => "bbcde"
```

Related: `String#getbyte`.

size()

Returns the count of characters (not bytes) in `self`:

```
'foo'.length        # => 3
'тест'.length       # => 4
'こんにちは'.length   # => 5
```

Contrast with `String#bytesize`:

```
'foo'.bytesize        # => 3
'тест'.bytesize       # => 8
'こんにちは'.bytesize   # => 15
```

Alias for: length

slice(*args)

Returns the substring of `self` specified by the arguments. See examples at String Slices.

Alias for: []

slice!(index) → new_string or nil
slice!(start, length) → new_string or nil
slice!(range) → new_string or nil
slice!(regexp, capture = 0) → new_string or nil
slice!(substring) → new_string or nil

Removes and returns the substring of `self` specified by the arguments. See String Slices.

A few examples:

```
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(field_sep = $;, limit = nil) → array
split(field_sep = $;, limit = nil) {|substring| ... } → self

Returns an array of substrings of `self` that are the result of splitting `self` at each occurrence of the given field separator `field_sep`.

When `field_sep` is `$;`:

- If `$;` is `nil` (its default value), the split occurs just as if `field_sep` were given as a space character (see below).
- If `$;` is a string, the split ocurs just as if `field_sep` were given as that string (see below).

When `field_sep` is `' '` and `limit` is `nil`, the split occurs at each sequence of whitespace:

```
'abc def ghi'.split(' ')          => ["abc", "def", "ghi"]
"abc \n\tdef\t\n  ghi".split(' ') # => ["abc", "def", "ghi"]
'abc  def   ghi'.split(' ')       => ["abc", "def", "ghi"]
''.split(' ')                     => []
```

When `field_sep` is a string different from `' '` and `limit` is `nil`, the split occurs at each occurrence of `field_sep`; trailing empty substrings are not returned:

```
'abracadabra'.split('ab')  => ["", "racad", "ra"]
'aaabcdaaa'.split('a')     => ["", "", "", "bcd"]
''.split('a')              => []
'3.14159'.split('1')       => ["3.", "4", "59"]
'!@#$%^$&*($)_+'.split('$') # => ["!@#", "%^", "&*(", ")_+"]
'тест'.split('т')          => ["", "ес"]
'こんにちは'.split('に')     => ["こん", "ちは"]
```

When `field_sep` is a Regexp and `limit` is `nil`, the split occurs at each occurrence of a match; trailing empty substrings are not returned:

```
'abracadabra'.split(/ab/) # => ["", "racad", "ra"]
'aaabcdaaa'.split(/a/)    => ["", "", "", "bcd"]
'aaabcdaaa'.split(//)     => ["a", "a", "a", "b", "c", "d", "a", "a", "a"]
'1 + 1 == 2'.split(/\W+/) # => ["1", "1", "2"]
```

If the Regexp contains groups, their matches are also included in the returned array:

```
'1:2:3'.split(/(:)()()/, 2) # => ["1", ":", "", "", "2:3"]
```

As seen above, if `limit` is `nil`, trailing empty substrings are not returned; the same is true if `limit` is zero:

```
'aaabcdaaa'.split('a')    => ["", "", "", "bcd"]
'aaabcdaaa'.split('a', 0) # => ["", "", "", "bcd"]
```

If `limit` is positive integer `n`, no more than `n - 1` splits occur, so that at most `n` substrings are returned, and trailing empty substrings are included:

```
'aaabcdaaa'.split('a', 1) # => ["aaabcdaaa"]
'aaabcdaaa'.split('a', 2) # => ["", "aabcdaaa"]
```

```
'aaabcdaaa'.split('a', 5) # => ["", "", "", "bcd", "aa"]
'aaabcdaaa'.split('a', 7) # => ["", "", "", "bcd", "", "", ""]
'aaabcdaaa'.split('a', 8) # => ["", "", "", "bcd", "", "", ""]
```

Note that if `field_sep` is a Regexp containing groups, their matches are in the returned array, but do not count toward the limit.

If `limit` is negative, it behaves the same as if `limit` was `nil`, meaning that there is no limit, and trailing empty substrings are included:

```
'aaabcdaaa'.split('a', -1) # => ["", "", "", "bcd", "", "", ""]
```

If a block is given, it is called with each substring:

```
'abc def ghi'.split(' ') {|substring| p substring }
```

Output:

```
"abc"
"def"
"ghi"
```

Related: String#partition, String#rpartition.

squeeze(*selectors) → new_string

Returns a copy of `self` with characters specified by `selectors` “squeezed” (see Multiple Character Selectors):

“Squeezed” means that each multiple-character run of a selected character is squeezed down to a single character; with no arguments given, squeezes all characters:

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

squeeze!(*selectors) → self or nil

Like String#squeeze, but modifies `self` in place. Returns `self` if any changes were made, `nil` otherwise.

start_with?(*string_or_regexp) → true or false

Returns whether `self` starts with any of the given `string_or_regexp`.

Matches patterns against the beginning of `self`. For each given `string_or_regexp`, the pattern is:

- `string_or_regexp` itself, if it is a `Regexp`.
- `Regexp.quote(string_or_regexp)`, if `string_or_regexp` is a string.

Returns `true` if any pattern matches the beginning, `false` otherwise:

```
'hello'.start_with?('hell')               # => true
'hello'.start_with?(/H/i)                 # => true
'hello'.start_with?('heaven', 'hell')     # => true
'hello'.start_with?('heaven', 'paradise') # => false
'тест'.start_with?('т')                   # => true
'こんにちは'.start_with?('こ')             # => true
```

Related: `String#end_with?`.

strip → new_string

Returns a copy of the receiver with leading and trailing whitespace removed; see Whitespace in Strings:

```
whitespace = "\x00\t\n\v\f\r "
s = whitespace + 'abc' + whitespace
s       # => "\u0000\t\n\v\f\r abc\u0000\t\n\v\f\r "
s.strip # => "abc"
```

Related: `String#lstrip`, `String#rstrip`.

strip! → self or nil

Like `String#strip`, except that any modifications are made in `self`; returns `self` if any modification are made, `nil` otherwise.

Related: `String#lstrip!`, `String#strip!`.

sub(pattern, replacement) → new_string
sub(pattern) {|match| ... } → new_string

Returns a copy of `self` with only the first occurrence (not all occurrences) of the given `pattern` replaced.

See Substitution Methods.

Related: `String#sub!`, `String#gsub`, `String#gsub!`.

sub!(pattern, replacement) → self or nil

sub!(pattern) {|match| ... } → self or nil

Returns `self` with only the first occurrence (not all occurrences) of the given `pattern` replaced.

See Substitution Methods.

Related: `String#sub`, `String#gsub`, `String#gsub!`.

succ → new_str

Returns the successor to `self`. The successor is calculated by incrementing characters.

The first character to be incremented is the rightmost alphanumeric: or, if no alphanumerics, the rightmost character:

```
'THX1138'.succ # => "THX1139"
'<<koala>>'.succ # => "<<koalb>>"
'***'.succ # => '**+'
```

The successor to a digit is another digit, “carrying” to the next-left character for a “rollover” from 9 to 0, and prepending another digit if necessary:

```
'00'.succ # => "01"
'09'.succ # => "10"
'99'.succ # => "100"
```

The successor to a letter is another letter of the same case, carrying to the next-left character for a rollover, and prepending another same-case letter if necessary:

```
'aa'.succ # => "ab"
'az'.succ # => "ba"
'zz'.succ # => "aaa"
'AA'.succ # => "AB"
'AZ'.succ # => "BA"
'ZZ'.succ # => "AAA"
```

The successor to a non-alphanumeric character is the next character in the underlying character set’s collating sequence, carrying to the next-left character for a rollover, and prepending another character if necessary:

```
s = 0.chr * 3
s # => "\x00\x00\x00"
s.succ # => "\x00\x00\x01"
s = 255.chr * 3
s # => "\xFF\xFF\xFF"
s.succ # => "\x01\x00\x00\x00"
```

Carrying can occur between and among mixtures of alphanumeric characters:

```
s = 'zz99zz99'
s.succ # => "aaa00aa00"
s = '99zz99zz'
s.succ # => "100aa00aa"
```

The successor to an empty String is a new empty String:

```
''.succ # => ""
```

Also aliased as: next

succ! → self

Equivalent to `String#succ`, but modifies `self` in place; returns `self`.

Also aliased as: next!

sum(n = 16) → integer

Returns a basic `n`-bit checksum of the characters in `self`; the checksum is the sum of the binary value of each byte in `self`, modulo `2**n - 1`:

```
'hello'.sum     # => 532
'hello'.sum(4)  # => 4
'hello'.sum(64) # => 532
'тест'.sum      # => 1405
'こんにちは'.sum  # => 2582
```

This is not a particularly strong checksum.

swapcase(*options) → string

Returns a string containing the characters in `self`, with cases reversed; each uppercase character is downcased; each lowercase character is upcased:

```
s = 'Hello World!' # => "Hello World!"
s.swapcase         # => "hELLO wORLD!"
```

The casing may be affected by the given `options`; see Case Mapping.

Related: `String#swapcase!`.

swapcase!(*options) → self or nil

Upcases each lowercase character in `self`; downcases uppercase character; returns `self` if any changes were made, `nil` otherwise:

```
s = 'Hello World!' # => "Hello World!"
s.swapcase!        # => "hELLO wORLD!"
s                  # => "hELLO wORLD!"
''.swapcase!       # => nil
```

The casing may be affected by the given `options`; see Case Mapping.

Related: `String#swapcase`.

to_c → complex

Returns a complex which denotes the string form. The parser ignores leading whitespaces and trailing garbage. Any digit sequences can be separated by an underscore. Returns zero for null or garbage string.

```
'9'.to_c           #=> (9+0i)
'2.5'.to_c         #=> (2.5+0i)
'2.5/1'.to_c       #=> ((5/2)+0i)
'-3/2'.to_c        #=> ((-3/2)+0i)
'-i'.to_c          #=> (0-1i)
'45i'.to_c         #=> (0+45i)
'3-4i'.to_c        #=> (3-4i)
'-4e2-4e-2i'.to_c  #=> (-400.0-0.04i)
'-0.0-0.0i'.to_c   #=> (-0.0-0.0i)
'1/2+3/4i'.to_c    #=> ((1/2)+(3/4)*i)
'ruby'.to_c        #=> (0+0i)
```

Polar form:

```
include Math
"1.0@0".to_c         #=> (1+0.0i)
"1.0@#{PI/2}".to_c   #=> (0.0+1i)
"1.0@#{PI}".to_c     #=> (-1+0.0i)
```

See `Kernel.Complex`.

to_f → float

Returns the result of interpreting leading characters in `self` as a Float:

```
'3.14159'.to_f  # => 3.14159
'1.234e-2'.to_f # => 0.01234
```

Characters past a leading valid number (in the given `base`) are ignored:

```
'3.14 (pi to two places)'.to_f # => 3.14
```

Returns zero if there is no leading valid number:

```
'abcdef'.to_f # => 0.0
```

to_i(base = 10) → integer

Returns the result of interpreting leading characters in `self` as an integer in the given `base` (which must be in (0, 2..36)):

```
'123456'.to_i      # => 123456
'123def'.to_i(16) # => 1195503
```

With `base` zero, string `object` may contain leading characters to specify the actual base:

```
'123def'.to_i(0)   # => 123
'0123def'.to_i(0)  # => 83
'0b123def'.to_i(0) # => 1
'0o123def'.to_i(0) # => 83
'0d123def'.to_i(0) # => 123
'0x123def'.to_i(0) # => 1195503
```

Characters past a leading valid number (in the given `base`) are ignored:

```
'12.345'.to_i   # => 12
'12345'.to_i(2) # => 1
```

Returns zero if there is no leading valid number:

```
'abcdef'.to_i # => 0
'2'.to_i(2)   # => 0
```

to_r → rational

Returns the result of interpreting leading characters in `str` as a rational. Leading whitespace and extraneous characters past the end of a valid number are ignored. Digit sequences can be separated by an underscore. If there is not a valid number at the start of `str`, zero is returned. This method never raises an exception.

```
'  2  '.to_r       #=> (2/1)
'300/2'.to_r       #=> (150/1)
'-9.2'.to_r        #=> (-46/5)
'-9.2e2'.to_r      #=> (-920/1)
```

```
'1_234_567'.to_r   #=> (1234567/1)
'21 June 09'.to_r  #=> (21/1)
'21/06/09'.to_r    #=> (7/2)
'BWV 1079'.to_r    #=> (0/1)
```

NOTE: “0.3”.to_r isn’t the same as 0.3.to_r. The former is equivalent to “3/10”.to_r, but the latter isn’t so.

```
"0.3".to_r == 3/10r  #=> true
0.3.to_r   == 3/10r  #=> false
```

See also `Kernel#Rational`.

to_s → self or string

Returns `self` if `self` is a String, or `self` converted to a String if `self` is a subclass of String.

Also aliased as: to_str

to_str()

Returns `self` if `self` is a String, or `self` converted to a String if `self` is a subclass of String.

Alias for: to_s

to_sym → symbol

Returns the `Symbol` corresponding to *str*, creating the symbol if it did not previously exist. See `Symbol#id2name`.

```
"Koala".intern         #=> :Koala
s = 'cat'.to_sym       #=> :cat
s == :cat              #=> true
s = '@cat'.to_sym      #=> :@cat
s == :@cat             #=> true
```

This can also be used to create symbols that cannot be represented using the `:xxx` notation.

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

Alias for: intern

tr(selector, replacements) → new_string

Returns a copy of `self` with each character specified by string `selector` translated to the corresponding character in string `replacements`. The correspondence is *positional*:

- Each occurrence of the first character specified by `selector` is translated to the first character in `replacements`.
- Each occurrence of the second character specified by `selector` is translated to the second character in `replacements`.
- And so on.

Example:

```
'hello'.tr('el', 'ip') #=> "hippo"
```

If `replacements` is shorter than `selector`, it is implicitly padded with its own last character:

```
'hello'.tr('aeiou', '-')   # => "h-ll-"
'hello'.tr('aeiou', 'AA-') # => "hAll-"
```

Arguments `selector` and `replacements` must be valid character selectors (see Character Selectors), and may use any of its valid forms, including negation, ranges, and escaping:

```
# Negation.
'hello'.tr('^aeiou', '-') # => "-e--o"
# Ranges.
'ibm'.tr('b-z', 'a-z') # => "hal"
# Escapes.
'hel^lo'.tr('\^aeiou', '-')     # => "h-l-l-"    # Escaped leading caret.
'i-b-m'.tr('b\-z', 'a-z')       # => "ibabm"     # Escaped embedded hyphen.
'foo\\bar'.tr('ab\\', 'XYZ')    # => "fooZYXr"   # Escaped backslash.
```

tr!(selector, replacements) → self or nil

Like `String#tr`, but modifies `self` in place. Returns `self` if any changes were made, `nil` otherwise.

tr_s(selector, replacements) → string

Like `String#tr`, but also squeezes the modified portions of the translated string; returns a new string (translated and squeezed).

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

```
'hello'.tr_s('el', 'hx') #=> "hhxo"
```

Related: String#squeeze.

tr_s!(selector, replacements) → self or nil

Like String#tr_s, but modifies self in place. Returns self if any changes were made, nil otherwise.

Related: String#squeeze!.

undump → string

Returns an unescaped version of self:

```
s_orig = "\f\x00\xff\\\""     # => "\f\u0000\xFF\\\""
s_dumped = s_orig.dump        # => "\"\\f\\x00\\xFF\\\\\\\"\""
s_undumped = s_dumped.undump # => "\f\u0000\xFF\\\""
s_undumped == s_orig          # => true
```

Related: String#dump (inverse of String#undump).

unicode_normalize(form = :nfc) → string

Returns a copy of self with Unicode normalization applied.

Argument form must be one of the following symbols (see Unicode normalization forms):

- :nfc: Canonical decomposition, followed by canonical composition.
- :nfd: Canonical decomposition.
- :nfkc: Compatibility decomposition, followed by canonical composition.
- :nfkd: Compatibility decomposition.

The encoding of self must be one of:

- Encoding::UTF_8
- Encoding::UTF_16BE
- Encoding::UTF_16LE
- Encoding::UTF_32BE
- Encoding::UTF_32LE
- Encoding::GB18030
- Encoding::UCS_2BE

- Encoding::UCS_4BE

Examples:

```
"a\u0300".unicode_normalize      # => "a"
"\u00E0".unicode_normalize(:nfd) # => "a "
```

Related: String#unicode_normalize!, String#unicode_normalized?.

unicode_normalize!(form = :nfc) → self

Like String#unicode_normalize, except that the normalization is performed on self.

Related String#unicode_normalized?.

unicode_normalized?(form = :nfc) → true or false

Returns true if self is in the given form of Unicode normalization, false otherwise. The form must be one of :nfc, :nfd, :nfkc, or :nfkd.

Examples:

```
"a\u0300".unicode_normalized?       # => false
"a\u0300".unicode_normalized?(:nfd) # => true
"\u00E0".unicode_normalized?        # => true
"\u00E0".unicode_normalized?(:nfd)  # => false
```

Raises an exception if self is not in a Unicode encoding:

```
s = "\xE0".force_encoding('ISO-8859-1')
s.unicode_normalized? # Raises Encoding::CompatibilityError.
```

Related: String#unicode_normalize, String#unicode_normalize!.

unpack(template, offset: 0) → array

Extracts data from self, forming objects that become the elements of a new array; returns that array. See Packed Data.

unpack1(template, offset: 0) → object

Like String#unpack, but unpacks and returns only the first extracted object. See Packed Data.

upcase(*options) → string

Returns a string containing the upcased characters in `self`:

```
s = 'Hello World!' # => "Hello World!"
s.upcase           # => "HELLO WORLD!"
```

The casing may be affected by the given `options`; see Case Mapping.

Related: `String#upcase!`, `String#downcase`, `String#downcase!`.

upcase!(*options) → self or nil

Upcases the characters in `self`; returns `self` if any changes were made, `nil` otherwise:

```
s = 'Hello World!' # => "Hello World!"
s.upcase!          # => "HELLO WORLD!"
s                  # => "HELLO WORLD!"
s.upcase!          # => nil
```

The casing may be affected by the given `options`; see Case Mapping.

Related: `String#upcase`, `String#downcase`, `String#downcase!`.

upto(other_string, exclusive = false) {|string| ... } → self
upto(other_string, exclusive = false) → new_enumerator

With a block given, calls the block with each String value returned by successive calls to `String#succ`; the first value is `self`, the next is `self.succ`, and so on; the sequence terminates when value `other_string` is reached; returns `self`:

```
'a8'.upto('b6') {|s| print s, ' ' } # => "a8"
```

Output:

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

If argument `exclusive` is given as a truthy object, the last value is omitted:

```
'a8'.upto('b6', true) {|s| print s, ' ' } # => "a8"
```

Output:

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

If `other_string` would not be reached, does not call the block:

```
'25'.upto('5') {|s| fail s }
'aa'.upto('a') {|s| fail s }
```

With no block given, returns a new Enumerator:

```
'a8'.upto('b6') # => #<Enumerator: "a8":upto("b6")>
```

valid_encoding? → true or false

Returns `true` if `self` is encoded correctly, `false` otherwise:

```
"\xc2\xa1".force_encoding("UTF-8").valid_encoding? # => true
"\xc2".force_encoding("UTF-8").valid_encoding?     # => false
"\x80".force_encoding("UTF-8").valid_encoding?     # => false
```