Ruby Language Reference Manual

version 1.4.6

Yukihiro Matsumoto
matz@zetabits.com

Table of Contents

- Preface
- Command-line options
- Ruby syntax
- The standard library
  - Built-in functions
  - Predefined variables
  - Predefined constants
  - Predefined classes
  - Predefined modules
- Bundled libraries
- **Ruby** Syntax in Pseudo BNF

Last modified: Mon Feb 23 16:01:41 1998

Preface

Ruby is the interpreted scripting language for quick and easy object-oriented programming. It has many features to process text files and to do system management tasks (as in Perl). It is simple, straight-forward, and extensible.

If you want a language for easy object-oriented programming, or you don't like the PERL ugliness, or you do like the concept of lisp, but don't like too much parentheses, Ruby may be the language of the choice.

Ruby's features are as follows:

Interpretive

Ruby is the interpreted language, so you don't have to recompile to execute the program written in Ruby.

Variables have no type (dynamic typing)

Variables in Ruby can contain data of any type. You don't have to worry about variable typing. Consequently, it has weaker compile time check.

No declaration needed

You can use variables in your Ruby programs without any declarations. Variable name itself denotes its scope (local, global, instance, etc.)

Simple syntax

Ruby has simple syntax influenced slightly from Eiffel.

No user-level memory management

Ruby has automatic memory management. Objects no longer referenced from anywhere are automatically collected by the garbage collector built in the interpreter.

Everything is object

Ruby is the pure object-oriented language from the beginning. Even basic data like integers are treated uniformly as objects.

Class, inheritance, methods

Of course, as a O-O language, Ruby has basic features like classes, inheritance, methods, etc.

Singleton methods

Ruby has the feature to define methods for certain specified object. For example, you can define a press-button action for certain GUI button by defining a singleton method for the button. Or, you can make up your own prototype based object system using singleton methods (if you want to).

Mix-in by modules

Ruby does not have the multiple inheritance intentionally. IMO, It is the source of confusion. Instead, Ruby has modules to share the implementation across the inheritance tree. It is often called the "Mix-in."

Iterators

Ruby has iterators for loop abstraction.

Closures

In Ruby, you can objectify the procedure.

Text processing and regular expression

Ruby has bunch of text processing features like in Perl.

Bignums

With built-in bignums, you can calculate factorial(400), for example.

Exception handling

As in Java(tm).

Direct access to OS

Ruby can call most of system calls on UNIX boxes. It can be used in system programming.

Dynamic loading

You can load object files into Ruby interpreter on-the-fly, on most of UNIXes.

Command line options

Ruby interpreter accepts following command-line options (switches). Basically they are quite similar to those of Perl.

`-0`*digit*
: specifies the input record separator (`$/`) as an octal number. If no digits given, the null character is the separator. Other switches may follow the digits. `-00` turns Ruby into paragraph mode. `-0777` makes Ruby read whole file at once as a single string, since there is no legal character with that value.

`-a`
: turns on auto-split mode when used with `-n` or `-p`. In auto-split mode, Ruby executes

```
$F = $_.split
```

at beginning of each loop.

`-c`
: causes Ruby to check the syntax of the script and exit without executing. If there is no syntax error, Ruby will print `"Syntax OK"` to the standard output.

`-K`*c*
: specifies KANJI (Japanese character) code-set.

`-d`
`--debug`
: turns on debug mode. `$DEBUG` will set true.

`-e` *script*
: specifies script from command-line. if `-e` switch specified, Ruby will not look for a script filename in the arguments.

`-F` *regexp*
: specifies input field separator (`$;`).

`-h`
`--help`
: prints a summary of the options.

`-i` *extension*
: specifies in-place-edit mode. The extension, if specified, is added to old filename to make a backup copy.

example:

```
% echo matz > /tmp/junk
% cat /tmp/junk
matz
% ruby -p -i.bak -e '$_.upcase!' /tmp/junk
% cat /tmp/junk
MATZ
% cat /tmp/junk.bak
matz
```

`-I` *directory*
: used to tell Ruby where to load the library scripts. Directory path will be added to the load-path variable (`$:').

`-l`
: enables automatic line-ending processing, which means firstly set `$\` to the value of `$/`, and secondly chops every line read using `chop!`, when used with `-n` or `-p`.

`-n`
: causes Ruby to assume the following loop around your script, which makes it iterate over filename arguments somewhat like `sed -n` or `awk`.

```
while gets
  ...
end
```

`-p`
: acts mostly same as `-n` switch, but print the value of variable `$_` at the each end of the loop.

example:

```
% echo matz | ruby -p -e '$_.tr! "a-z", "A-Z"'
MATZ
```

`-r filename`
: causes Ruby to load the file using `require`. It is useful with switches `-n` or `-p`.

`-s`
: enables some switch parsing for switches after script name but before any filename arguments (or before a `--`). Any switches found there is removed from ARGV and set the corresponding variable in the script.

example:

```
#! /usr/local/bin/ruby -s
# prints "true" if invoked with `-xyz' switch.
print "true\n" if $xyz
```

`-S`
: makes Ruby uses the PATH environment variable to search for script, unless if its name begins with a slash. This is used to emulate `#!` on machines that don't support it, in the following manner:

```
#!/bin/sh
exec ruby -S -x $0 "$@"
#! ruby
```

On some systems $0 does not always contain the full pathname, so you need `-S` switch to tell Ruby to search for the script if necessary.

`-T [level]`
: Forces "taint" checks to be turned on so you can test them. If level is specified, `$SAFE` to be set to that level. It's a good idea to turn them on explicitly for programs run on another's behalf, such as CGI programs.

`-v`
`--verbose`
: enables verbose mode. Ruby will prints its version at the beginning, and set the variable `` `$VERBOSE' `` to true. Some methods prints extra messages if this variable is true. If this switch is given, and no other switches present, Ruby quits after printing its version.

`--version`
: prints the version of Ruby executable.

`-w`
: enables verbose mode without printing version message at the beginning. It set the variable `` `$VERBOSE' `` to true.

`-x[directory]`
: tells Ruby that the script is embedded in a message. Leading garbage will be discarded until the first that starts with "#!" and contains string "ruby". Any meaningful switches on that line will applied. The end of script must be specified with either EOF, `^D` (control-D), `^Z` (control-Z), or reserved word `__END__`.If the directory name is specified, Ruby will switch to that directory before executing script.

`-X directory`
: causes Ruby to switch to the directory.

`-y`
`--yydebug`
: turns on compiler debug mode. Ruby will print bunch of internal state messages during compiling scripts. You don't have to specify this switch, unless you are going to debug the Ruby interpreter itself.

Ruby syntax

- Lexical structure
  - Identifiers
  - Comment
  - Embedded Documentation
  - Reserved words
- Program

- Expressions
  - String literals
  - Command output
  - Regular expressions
  - Expression substitution in strings
  - line-oriented string literals (Here document)
  - Numeric literals
  - Variables and constants
  - Global variables
  - Instance variables
  - Local variables
  - Pseudo variables
  - Constants
  - Array expressions
  - Hash expressions
  - Method invocation
  - `super`
  - Assignment
  - Operator expressions
  - Control structure
    - `if`
    - `if` modifier
    - `unless`
    - `unless` modifier
    - `case`
    - `and`
    - `or`
    - `not`
    - Range expressions
    - `while`
    - `while` modifier
    - `until`
    - `until` modifier
    - Iterators
    - `for`
    - `yield`
    - `raise`
    - `begin`
    - `retry`
    - `return`
    - `break`
    - `next`
    - `redo`
    - BEGIN
    - END
  - Class definitions
  - Singleton-class definitions
  - Module definitions
  - Method definitions
  - Singleton-method definitions
  - `alias`
  - `undef`
  - `defined?`

Lexical structure

The character set used in the Ruby source files for the current implementation is based on ASCII. The case of characters in source files is significant. All syntactic constructs except identifiers and certain literals may be separated by an arbitrary number of whitespace characters and comments. The whitespace characters are space, tab, vertical tab, backspace, carriage return, and form feed. Newlines works as whitespace only when expressions obviously continues to the next line.

Identifiers

Examples:

```
foobar
ruby_is_simple
```

Ruby identifiers are consist of alphabets, decimal digits, and the underscore character, and begin with a alphabets (including underscore). There are no restrictions on the lengths of Ruby identifiers.

Comment

Examples:

```
# this is a comment line
```

Ruby comments start with "#" outside of a string or character literal (`?#`) and all following text until the end of the line.

Embedded Documentation

Example:

```
=begin
the everything between a line beginning with `=begin' and
that with `=end' will be skipped by the interpreter.
=end
```

If the Ruby interpreter encounters a line beginning with `=begin`, it skips that line and all remaining lines through and including a line that begins with `=end`.

Reserved words

The reserved words are:

```
BEGIN     class     ensure   nil        self     when
END       def       false    not        super    while
alias     defined   for        or       then     yield
and       do        if         redo     true
begin     else      in         rescue   undef
break     elsif     module   retry    unless
case      end       next       return   until
```

Program

Example:

```
print "hello world!\n"
```

Ruby programs are sequence of expressions. Each expression are delimited by semicolons(`;`) or newlines. Backslashes at the end of line does not terminate expression.

Expression

Examples:

```
true
(1+2)*3
```

```
foo()
if test then ok else ng end
```

Ruby expressions can be grouped by parentheses.

String literals

Examples:

```
"this is a string expression\n"
"concat#{foobar}"
'concat#{foobar}'
%q!I said, "You said, 'She said it.'"!
%!I said, "You said, 'She said it.'"!
%Q('This is it.'\n)
```

String expressions begin and end with double or single quote marks. Double-quoted string expressions are subject to backslash escape and expression substitution. Single-quoted strings are not (except for `\'` and `\\`).

The string expressions begin with `%` are the special form to avoid putting too many backslashes into quoted strings. The `%q/STRING/` expression is the generalized single quote. The `%Q/STRING/` (or `%/STRING/`) expression is the generalized double quote. Any non-alphanumeric delimiter can be used in place of `/`, including newline. If the delimiter is an opening bracket or parenthesis, the final delimiter will be the corresponding closing bracket or parenthesis. (Embedded occurrences of the closing bracket need to be backslashed as usual.)

Backslash notation

`\t`	tab(0x09)
`\n`	newline(0x0a)
`\r`	carriage return(0x0d)
`\f`	form feed(0x0c)
`\b`	backspace(0x08)
`\a`	bell(0x07)
`\e`	escape(0x1b)
`\s`	whitespace(0x20)
`\nnn`	character in octal value nnn
`\xnn`	character in hexadecimal value nn
`\cx`	control x
`\C-x`	control x
`\M-x`	meta x (c | 0x80)
`\M-\C-x`	meta control x
`\x`	character x itself

The string literal expression yields new string object each time it evaluated.

Command output

Examples:

```
`date`
%x{ date }
```

Strings delimited by backquotes are performed by a subshell after escape sequences interpretation and expression substitution. The standard output from the commands are taken as the value. Commands performed each time they evaluated.

The `%x/STRING/` is the another form of the command output expression.

Regular expression

Examples:

```
/^Ruby the OOPL/
/Ruby/i
/my name is #{myname}/o
%r|^/usr/local/.*|
```

Strings delimited by slashes are regular expressions. The characters right after latter slash denotes the option to the regular expression. Option `i` means that regular expression is case insensitive. Option `i` means that regular expression does expression substitution only once at the first time it evaluated. Option `x` means extended regular expression, which means whitespaces and commens are allowd in the expression. Option `p` denotes POSIX mode, in which newlines are treated as normal character (matches with dots).

The `%r/STRING/` is the another form of the regular expression.

`^`	beginning of a line or string
`$`	end of a line or string
`.`	any character except newline
`\w`	word character[0-9A-Za-z_]
`\W`	non-word character
`\s`	whitespace character[\t\n\r\f]
`\S`	non-whitespace character
`\d`	digit, same as[0-9]
`\D`	non-digit
`\A`	beginning of a string
`\Z`	end of a string, or before newline at the end
`\z`	end of a string
`\b`	word boundary(outside[]only)
`\B`	non-word boundary
`\b`	backspace(0x08)(inside[]only)
`[ ]`	any single character of set
`*`	0 or more previous regular expression
`*?`	0 or more previous regular expression(non greedy)
`+`	1 or more previous regular expression
`+?`	1 or more previous regular expression(non greedy)
`{m,n}`	at least m but most n previous regular expression
`{m,n}?`	at least m but most n previous regular expression(non greedy)
`?`	0 or 1 previous regular expression
`|`	alternation
`( )`	grouping regular expressions
`(?# )`	comment
`(?: )`	grouping without backreferences
`(?= )`	zero-width positive look-ahead assertion
`(?! )`	zero-width negative look-ahead assertion
`(?ix-ix)`	turns on (or off) `i' and `x' options within regular expression. These modifiers are localized inside an enclosing group (if any).
`(?ix-ix: )`	turns on (or off) `i' and `x' options within this non-capturing group.

Backslash notation and expression substitution available in regular expressions.

Expression substitution in strings

Examples:

```
"my name is #{$ruby}"
```

In double-quoted strings, regular expressions, and command output expressions, the form like "#{expression}" extended to the evaluated result of that expression. If the expressions are the variables which names begin with the

character either `$',`@', expressions are not needed to be surrounded by braces. The character `#' is interpreted literally if it it not followed by characters `{',`$',`@'.

line-oriented string literals (Here document)

There's a line-oriente form of the string literals that is usually called as `here document'. Following a << you can specify a string or an identifier to terminate the string literal, and all lines following the current line up to the terminator are the value of the string. If the terminator is quoted, the type of quotes determines the type of the line-oriented string literal. Notice there must be no space between << and the terminator.

If the - placed before the delimiter, then all leading whitespcae characters (tabs or spaces) are stripped from input lines and the line containing delimiter. This allows here-documents within scripts to be indented in a natural fashion.

```
  print <<EOF
The price is #{$Price}.
EOF

  print <<"EOF";                    # same as above
The price is #{$Price}.
EOF

  print <<`EOC`                     # execute commands
echo hi there
echo lo there
EOC

  print <<"foo", <<"bar"   # you can stack them
I said foo.
foo
I said bar.
bar

  myfunc(<<"THIS", 23, <<'THAT')
Here's a line
or two.
THIS
and here's another.
THAT

  if need_define_foo
    eval <<-EOS                     # delimiters can be indented
     def foo
       print "foo\n"
     end
    EOS
 end
```

Numeric literals

`123` integer
`-123` integer(signed)
`1_234` integer(underscore within decimal numbers ignored)
`123.45`
floating point number
`1.2e-3`
floating point number
`0xffff`
hexadecimal integer
`0b01011`
binary integer
`0377` octal integer
`?a` ASCII code for character `a'(97)
`?\C-a` Control-a(1)
`?\M-a` Meta-a(225)
`?\M-\C-a`
Meta-Control-a(129)

`:symbol`
: Integer corresponding identifiers, variable names, and operators.

In ?-representation all backslash notations are available.

Variables and constants

The variable in Ruby programs can be distinguished by the first character of its name. They are either global variables, instance variables, local variables, and class constants. There are no restriction for variable name length (except heap size).

Global variables

Examples:

```
$foobar
$/
```

The variable which name begins with the character `$', has global scope, and can be accessed from any location of the program. Global variables are available as long as the program lives. Non-initialized global variables has value `nil`.

Instance variables

Examples:

```
@foobar
```

The variable which name begins which the character `@', is an instance variable of `self`. Instance variables are belong to the certain object. Non-initialized instance variables has value `nil`.

Constants

Examples:

```
FOOBAR
```

The identifier which name begins with upper case letters ([A-Z]) is an constant. The constant definitions are done by assignment in the class definition body. Assignment to the constants must be done once. Changing the constant value or accessing to the non-initialized constants raises a `NameError` exception.

The constants can be accessed from:

- the class or module body in which the constant is defined, including the method body and the nested module/class definition body.
- the class which inherit the constant defining class.
- the class or module which includes the constant defining module.

Class definition defines the constant automatically, all class names are constants.

To access constants defined in certain class/module, operator `::` can be used.

To access constants defined in the Object class, operator `::` without the left hand side operand can be used.

Examples:

```
Foo::Bar
::Bar
```

No assignment using operator `::' is permitted.

Local variables

Examples:

```
foobar
```

The identifier which name begins with lower case character or underscore, is a local variable or a method invocation. The first assignment in the local scope (bodies of class, module, method definition) to such identifiers are declarations of the local variables. Non-declared identifiers are method invocation without arguments.

The local variables assigned first time in the blocks are only valid in that block. They are called `dynamic variables.' For example:

```
i0 = 1
loop {
  i1 = 2
  print defined?(i0), "\n" # true
  print defined?(i1), "\n" # true
  break
}
print defined?(i0), "\n"   # true
print defined?(i1), "\n"   # false
```

Pseudo variables

There are special variables called `pseudo variables'.

`self` the receiver of the current method
`nil` the sole instance of the Class NilClass(represents false)
`true` the sole instance of the Class TrueClass(typical true value)
`false` the sole instance of the Class FalseClass(represents false)
`__FILE__`
: the current source file name.
`__LINE__`
: the current line number in the source file.

The values of the pseudo variables cannot be changed. Assignment to these variables causes exceptions.

Array expression

Examples:

```
[1, 2, 3]
```

Syntax:

```
`[' expr,...`]'
```

Returns an array, which contains result of each expressions. Arrays are instances of the class Array.

`%w` expressions make creation of the arrays of strings easier. They are equivalent to the single quoted strings split by the whitespaces. For example:

```
%w(foo bar baz)
```

is equivalent to `["foo", "bar", "baz"]`. Note that parenthesis right after `%s` is the quote delimiter, not usual parenthesis.

Hash expression

Examples:

```
{1=>2, 2=>4, 3=>6}
```

Syntax:

```
{ expr => expr...}
```

Returns a new Hash object, which maps each key to corresponding value. Hashes are instances of the class Hash.

Method invocation

Examples:

```
foo.bar()
foo.bar
bar()
print "hello world\n"
print
```

Syntax:

```
[expr `.'] identifier [`(' expr...[`*' [expr]],[`&' ] expr`)']
[expr `::'] identifier [`(' expr...[`*' [expr]],[`&' expr] `)']
```

Method invocation expression invokes the method of the receiver (right hand side expression of the dot) specified by the identifier. If no receiver specified, `self` is used as a receiver.

Identifier names are normal identifiers and identifier suffixed by character `?` or `!`. As a convention, `identifier?` are used as predicate names, and `identifier!` are used for the more destructive (or more dangerous) methods than the method which have same name without `!`.

If the last argument expression preceded by `*`, the value of the expression expanded to arguments, that means

```
foo(*[1,2,3])
```

equals

```
foo(1,2,3)
```

If the last argument expression preceded by `&`, the value of the expression, which must be a `Proc` object, is set as the block for the calling method.

Some methods are *private*, and can be called from function form invocations (the forms that omits receiver).

`super`

Examples:

```
super
super(1,2,3)
```

Syntax:

```
super
super(expr,...)
```

the `super` invokes the method which the current method overrides. If no arguments given, arguments to the current method passed to the method.

Assignment

Examples:

```
foo = bar
foo[0] = bar
foo.bar = baz
```

Syntax:

```
variable '=' expr
constant '=' expr
expr`['expr..`]' '=' expr
expr`.'identifier '=' expr
```

Assignment expression are used to assign objects to the variables or such. Assignments sometimes work as declarations for local variables or class constants. The left hand side of the assignment expressions can be either:

- variables

  ```
  variables `=' expression
  ```

  If the left hand side is a variables, then assignment is directly performed.

- array reference

  ```
  expr1`[' expr2...`]' `=' exprN
  ```

  This from is evaluated to the invocation of the method named `[]=`, with expr1 as the receiver, and values expr2 to exprN as arguments.

- attribute reference

  ```
  expr `.' identifier `=' expr
  ```

  This from is evaluated to the invocation of the method named `identifier=` with the right hand side expression as a argument.

self assignment

Examples:

```
foo += 12
```

Syntax:

```
expr op= expr     # left hand side must be assignable.
```

This form evaluated as `expr = expr op expr`. But right hand side expression evaluated once. op can be one of:

```
+, -, *, /, %, **, &, |, ^, <<, >>, &&, ||
```

There may be no space between operators and `=`.

Multiple assignment

Examples:

```
foo, bar, baz = 1, 2, 3
foo, = list()
foo, *rest = list2()
```

Syntax:

```
expr `,' [expr `,'...] [`*' expr] = expr [, expr...][`*' [expr]]
`*' expr = expr [, expr...][`*' expr]
```

Multiple assignment form performs multiple assignment from expressions or an array. Each left hand side expression must be assignable. If single right hand side expression given, the value of the expression converted into an array, then each element in array assigned one by one to the left hand side expressions. If number of elements in the array is greater than left hand sides, they are just ignored. If left hand sides are longer than the array, `nil` will be added to the locations.

Multiple assignment acts like this:

```
foo, bar = [1, 2] # foo = 1; bar = 2
foo, bar = 1, 2           # foo = 1; bar = 2
foo, bar = 1              # foo = 1; bar = nil

foo, bar, baz = 1, 2      # foo = 1; bar = 2; baz = nil
foo, bar = 1, 2, 3        # foo = 1; bar = 2
foo,*bar = 1, 2, 3        # foo = 1; bar = [2, 3]
```

The value of the multiple assignment expressions are the array used to assign.

Operator expressions

Examples:

```
1+2*3/4
```

As a syntax sugar, several methods and control structures has operator form. Ruby has operators show below:

```
high   ::
       []
       **
       -(unary)  +(unary)  !  ~
       *  /  %
       +  -
       <<  >>
       &
       |  ^
       >  >=  <  <=
       <=> ==  === !=  =~  !~
       &&
       ||
       .. ...
       =(+=, -=...)
       not
low    and or
```

Most of operators are just method invocation in special form. But some operators are not methods, but built in to the syntax:

```
=, .., ..., !, not, &&, and, ||, or, !=, !~
```

In addition, assignment operators(+= etc.) are not user-definable.

Control structure

Control structures in Ruby are expressions, and have some value. Ruby has the loop abstraction feature called iterators. Iterators are user-definable loop structure.

`if`

Examples:

```
if age >= 12 then
  print "adult fee\n"
else
  print "child fee\n"
end
gender = if foo.gender == "male" then "male" else "female" end
```

Syntax:

```
if expr [then]
  expr...
[elsif expr [then]
  expr...]...
[else
  expr...]
end
```

`if` expressions are used for conditional execution. The values `false` and `nil` are false, and everything else are true. Notice Ruby uses `elsif`, not `else if` nor `elif`.

If conditional part of `if` is the regular expression literal, then it evaluated like:

```
$_ =~ /re/
```

`if` modifier

Examples:

```
print "debug\n" if $debug
```

Syntax:

```
expr if expr
```

executes left hand side expression, if right hand side expression is true.

`unless`

Examples:

```
unless $baby
  feed_meat
else
  feed_milk
end
```

Syntax:

```
unless expr [then]
  expr...
[else
  expr...]
end
```

`unless` expressions are used for reverse conditional execution. It is equivalent to:

```
if !(cond)
  ...
else
  ...
end
```

`unless` modifier

Examples:

```
print "stop\n" unless valid($passwd)
```

Syntax:

```
expr unless expr
```

executes left hand side expression, if right hand side expression is false.

`case`

Examples:

```
case $age
when 0 .. 2
  "baby"
when 3 .. 6
  "little child"
when 7 .. 12
  "child"
when 12 .. 18
  # Note: 12 already matched by "child"
  "youth"
else
  "adult"
end
```

Syntax:

```
case expr
[when expr [, expr]...[then]
  expr..]..
[else
  expr..]
end
```

the `case` expressions are also for conditional execution. Comparisons are done by operator `===`. Thus:

```
case expr0
when expr1, expr2
  stmt1
when expr3, expr4
  stmt2
else
  stmt3
end
```

is basically same to below:

```
_tmp = expr0
if expr1 === _tmp || expr2 === _tmp
  stmt1
elsif expr3 === _tmp || expr4 === _tmp
  stmt2
else
  stmt3
end
```

Behavior of the `===` method varies for each Object. See docutmentation for each class.

`and`

Examples:

```
test && set
test and set
```

Syntax:

```
expr `&&' expr
expr `and' expr
```

Evaluates left hand side, then if the result is true, evaluates right hand side. `and` is lower precedence alias.

`or`

Examples:

```
demo || die
demo or die
```

Syntax:

```
expr `||' expr
expr or expr
```

Evaluates left hand side, then if the result is false, evaluates right hand side. `or` is lower precedence alias.

`not`

Examples:

```
! me
not me
i != you
```

Syntax:

```
`!' expr
not expr
```

Returns true if false, false if true.

```
expr `!=' expr
```

Syntax sugar for `!(expr == expr)`.

```
expr `!~' expr
```

Syntax sugar for `!(expr =~ expr)`.

Range expressions

Examples:

```
1 .. 20
/first/ ... /second/
```

Syntax:

```
expr `..' expr
expr `...' expr
```

If range expression appears in any other place than conditional expression, it returns range object from left hand side to right hand side.

If range expression appears in conditional expression, it gives false until left hand side returns true, it stays true until right hand side is true. `..` acts like `awk`, `...` acts like `sed`.

`while`

Examples:

```
while sunshine
  work()
end
```

Syntax:

```
while expr [do]
  ...
end
```

Executes body while condition expression returns true.

`while` modifier

Examples:

```
sleep while idle
```

Syntax:

```
expr while expr
```

Repeats evaluation of left hand side expression, while right hand side is true. If left hand side is `begin` expression, `while` evaluates that expression at lease once.

`until`

Examples:

```
until sunrise
  sleep
end
```

Syntax:

```
until expr [do]
  ...
end
```

Executes body until condition expression returns true.

`until` modifier

Examples:

```
work until tired
```

Syntax:

```
expr until expr
```

Repeats evaluation of left hand side expression, until right hand side is true. If left hand side is `begin` expression, `until` evaluates that expression at lease once.

Iterators

Examples:

```
[1,2,3].each do |i| print i*2, "\n" end
[1,2,3].each{|i| print i*2, "\n"}
```

Syntax:

```
method_call do [`|' expr...`|'] expr...end
method_call `{' [`|' expr...`|'] expr...`}'
```

The method may be invoked with the block (`do .. end` or `{..}`). The method may be evaluate back that block from inside of the invocation. The methods that calls back the blocks are sometimes called as iterators. The evaluation of the block from iterator is done by `yield`.

The difference between `do` and braces are:

- Braces has stronger precedence. For example:

```
foobar a, b do .. end      # foobar will be called with the block.
foobar a, b { .. }         # b will be called with the block.
```

- Braces introduce the nested local scopes, that is newly declared local variables in the braces are valid only in the blocks. For example:

```
foobar {
  i = 20          # local variable `i' declared in the block.
  ...
}
print defined? i  # `i' is not defined here.
foobar a, b { .. }         # it is not valid outside of the block
```

`for`

Examples:

```
for i in [1, 2, 3]
  print i*2, "\n"
end
```

Syntax:

```
for lhs... in expr [do]
  expr..
end
```

Executes body for each element in the result of expression. `for` is the syntax sugar for:

```
(expr).each `{' `|' lhs..`|' expr.. `}'
```

`yield`

Examples:

```
yield data
```

Syntax:

```
yield `(' [expr [`,' expr...]])
yield [expr [`,' expr...]]
```

Evaluates the block given to the current method with arguments, if no argument is given, `nil` is used as an argument. The argument assignment to the block prameter is done just like multiple assignment. If the block is not supplied for the current method, the exception is raised.

`raise`

Examples:

```
raise "you lose"  # raise RuntimeError
# both raises SyntaxError
raise SyntaxError, "invalid syntax"
raise SyntaxError.new("invalid syntax")
raise              # re-raise last exception
```

Syntax:

```
raise
raise message_or_exception
raise error_type, message
raise error_type, message, traceback
```

Raises a exception. In the first form, re-raises last exception. In second form, if the argument is the string, creates a new `RuntimeError` exception, and raises it. If the argument is the exception, `raise` raises it. In the third form, `raise` creates a new exception of type *error_type*, and raises it. In the last form, the third argument is the traceback information for the raising exception in the format given by variable *$@* or `caller` function.

The exception is assigned to the variable `$!`, and the position in the source file is assigned to the `$@`.

The word `raise' is not the reserved word in Ruby. `raise` is the method of the `Kernel` module. There is an alias named `fail`.

`begin`

Examples:

```
begin
  do_something
rescue
  recover
ensure
  must_to_do
end
```

Syntax:

```
begin
  expr..
[rescue [error_type,..]
  expr..]..
[else
  expr..]
[ensure
  expr..]
end
```

`begin` expression executes its body and returns the value of the last evaluated expression.

If an exception occurs in the `begin` body, the `rescue` clause with the matching exception type is executed (if any). The match is done by the `kind_of?`. The default value of the rescue clause argument is the `StandardError`, which is the superclass of most built-in exceptions. Non-local jumps like `SystemExit` or `Interrupt` are not subclass of the `StandardError`.

The `begin` statement has an optional `else` clause, which must follow all `rescue` clauses. It is executed if the `begin` body does not raise any exception.

For the `rescue` clauses, the `error_type` is evaluated just like the arguments to the method call, and the clause matches if the value of the variable *$!* is the instance of any one of the `error_type` of its subclass. If *error_type* is not class nor module, the `rescue` clause raises *TypeError* exception.

If `ensure` clause given, its clause body executed whenever `begin`body exits.

`retry`

Examples:

```
retry
```

Syntax:

```
retry
```

If `retry` appears in `rescue` clause of `begin` expression, restart from the beginning of the 1`begin` body.

```
begin
  do_something # exception raised
rescue
  # handles error
  retry  # restart from beginning
end
```

If `retry` appears in the iterator, the block, or the body of the `for` expression, restarts the invocation of the iterator call. Arguments to the iterator is re-evaluated.

```
for i in 1..5
  retry if some_condition # restart from i == 1
end

# user defined "until loop"
def UNTIL(cond)
  yield
  retry if not cond
end
```

`retry` out of `rescue` clause or iterators raises exception.

`return`

Examples:

```
return
return 12
return 1,2,3
```

Syntax:

```
return [expr[`,' expr...]]
```

Exits from method with the return value. If more than two expressions are given, the array contains these values will be the return value. If no expression given, `nil` will be the return value.

`break`

Examples:

```
i=0
while i<3
  print i, "\n"
  break
end
```

Syntax:

```
break
```

Exits from the most internal loop. Notice `break` does not exit from `case` expression like C.

`next`

Examples:

```
next
```

Syntax:

```
next
```

Jumps to next iteration of the most internal loop.

`redo`

Examples:

```
redo
```

Syntax:

```
redo
```

Restarts this iteration of the most internal loop, without checking loop condition.

BEGIN

Examples:

```
BEGIN {
  ...
}
```

Syntax:

```
BEGIN '{'
  expr..
'}'
```

Registers the initialize routine. The block followed after `BEGIN` is evaluated before any other statement in that file (or string). If multiple `BEGIN` blocks are given, they are evaluated in the appearing order.

The BEGIN block introduce new local-variable scope. They don't share local variables with outer statements.

The BEGIN statement can only appear at the toplevel.

END

Examples:

```
END {
  ...
}
```

Syntax:

```
END '{' expr.. '}'
```

Registers finalize routine. The block followed after END is evaluated just before the interpreter termination. Unlike BEGIN, END blocks shares their local variables, just like blocks.

The END statement registers its block only once at the first execution. If you want to register finalize routines many times, use at_exit.

The END statement can only appear at the toplevel. Also you cannot cancel finalize routine registered by END.

Class definitions

Examples:

```
class Foo < Super
  def test
     :
  end
     :
end
```

Syntax:

```
class identifier [`<' superclass ]
  expr..
end
```

Defines the new class. The class names are identifiers begin with uppercase character.

Singleton-class definitions

Examples:

```
class << obj
  def test
     :
  end
     :
end
```

Syntax:

```
class `<<' expr
  expr..
end
```

Defines the class attribute for certain object. The definitions within this syntax only affect the specified object.

Module definitions

Examples:

```
module Foo
  def test
     :
  end
     :
end
```

Syntax:

```
module identifier
  expr..
end
```

Defines the new module The module names are identifiers begin with uppercase character.

Method definitions

Examples:

```
def fact(n)
  if n == 1 then
     1
  else
    n * fact(n-1)
  end
end
```

Syntax:

```
def method_name [`(' [arg ['=' default]]...[`,' `*' arg ]`)']
  expr..
end
```

Defines the new method. Method_name should be either identifier or re-definable operators (e.g. ==, +, -, etc.). Notice the method is not available before the definition. For example:

```
foo
def foo
  print "foo\n"
end
```

will raise an exception for undefined method invoking.

The argument with default expression is optional. The evaluation of the default expression is done at the method invocation time. If the last argument preceded by `*`, actual parameters which don't have corresponding formal arguments are assigned in this argument as an array.

If the last argument preceded by `&`, the block given to the method is converted into the `Proc` object, and assigned in this argument. In case both `*` and `&` are present in the argument list, `&` should come later.

The method definitions can not be nested.

The return value of the method is the value given to the `return`, or that of the last evaluated expression.

Some methods are marked as `private', and must be called in the function form.

When the method is defined outside of the class definition, the method is marked as private by default. On the other hand, the methods defined in the class definition are marked as public by default. The default visibility and the `private' mark of the methods can be changed by `public` or `private` of the `Module`.

In addition, the methods named `initialize` are always defined as private methods.

Singleton-method definitions

Examples:

```
def foo.test
  print "this is foo\n"
end
```

Syntax:

```
def expr `.' identifier [`(' [arg [`=' default]]...[`,' `*' arg ]`)']
  expr..
end
```

The singleton-method is the method which belongs to certain object. The singleton-method definitions can be nested.

The singleton-methods of classes inherited to its subclasses. The singleton-methods of classes are acts like class methods in other object-oriented languages.

`alias`

Examples:

```
alias foo bar
alias $MATCH $&
```

Syntax:

```
alias method-name method-name
alias global-variable-name global-variable-name
```

Gives alias to methods or global variables. Aliases can not be defined within the method body.

The aliase of the method keep the current definition of the method, even when methods are overridden.

Making aliases for the numbered global variables (`$1`, `$2`,...) is prohibited. Overriding the builtin global variables may cause serious problems.

`undef`

Examples:

```
undef bar
```

Syntax:

```
undef method-name
```

Cancels the method definition. `Undef` can not appear in the method body. By using `undef` and `alias`, the interface of the class can be modified independently from the superclass, but notice it may be broke programs by the internal method call to self.

defined?

Examples:

```
defined? print
defined? File.print
defined?(foobar)
defined?($foobar)
defined?(@foobar)
defined?(Foobar)
```

Syntax:

```
defined? expr
```

Returns false if the expression is not defined. Returns the string that describes a kind of the expression.

Built-in functions

- `
- Array
- Float
- Integer
- String
- at_exit
- autoload
- binding
- caller
- catch
- chop
- chop!
- chomp
- chomp!
- eval
- exec
- exit
- exit!
- fail
- fork
- format
- gets
- global_variables
- gsub
- gsub!
- iterator?
- lambda
- load
- local_variables
- loop
- open
- p
- print
- printf

- proc
- putc
- puts
- raise
- rand
- readline
- readlines
- require
- select
- sleep
- split
- sprintf
- srand
- sub
- sub!
- syscall
- system
- test
- trace_var
- trap
- untrace_var

---

pre-defined functions

Some methods defined in the Kernel module can be called from everywhere, and are to be called like functions. You'd better think twice before redefining these methods.

`` ` str ``

Performs *str* by a subshell. The standard output from the commands are taken as the value. This method is called by a syntax sugar form like `` `str` ``.

`Array(arg)`

Converts the argument to the array using to_a.

`Float(arg)`

Converts the argument to the float value.

`Integer(arg)`

Converts the argument to the integer value. If the argument is string, and happen to start with 0x, 0b, 0, interprets it as hex, binary, octal string respectively.

`String(arg)`

Converts the argument to the string using Kernel#to_s.

`at_exit`

Register the block for clean-up to execute at the interpreter termination.

`autoload(module, file)`

Specifies *file* to be loaded using the method require, when *module* accessed for the first time. *module* must be a string or a symbol.

`binding`

Returns the data structure of the variable/method binding, which can be used for the second argument of the `eval`.

`caller([`*`level`*`])`

Returns the context information (the backtrace) of current call in the form used for the variable `$@`. When *level* specified, `caller` goes up to calling frames *level* times and returns the context information. `caller` returns an empty array at toplevel.

The lines below prints stack frame:

```
for c in caller(0)
  print c, "\n"
end
```

`catch(`*`tag`*`){...}`

Executes the block, and if an non-local exit named *tag* submitted by the *throw*, it returns with the value given by the `throw`.

For example, the code below returns the value 25, not 10, and the *some_process* never be called.

```
def throw_exit
  throw :exit, 25
end

catch(:exit) {
   throw_exit
   some_process;
   10;
}
```

`chop`
`chop!`

Removes off the last character of the value of the variable `$_` (2 characters if the last characters are "\r\n"). `chop!` modifies the string itself. `chop` makes a copy to modify.

`chomp([`*`rs`*`])`
`chomp!([`*`rs`*`])`

Removes off the line ending from the value of the variable `$_`. See String#chomp.

`eval(`*`expr`*`[, `*`binding`*`])`

Evaluate *expr* as a Ruby program. If the `Proc` object or the binding data from `binding` is given to the optional second argument, the string is compiled and evaluated under its binding environment.

`exec(`*`command`*`...)`

Executes *command* as a subprocess, and **never returns**.

If multiple arguments are given, `exec` invokes command directly, so that whitespaces and shell's meta-characters are not processed by the shell.

If the first argument is an array that has two elements, the first element is the real path for the command, and the second element is for the argv[0] to `execl(2)`.

`exit([status])`

Exits immediately with `status`. if *status* is omitted, exits with 0 status.

`exit` raises `SystemExit` to terminate the program, which can be handled by the `rescue` clause of the `begin` statement.

`exit!([status])`

Exits with status. Unlike `exit`, it ignores any kind of exception handling (including `ensure`). Used to terminate sub-process after calling fork.

`fork`

Does a `fork(2)` system call. Returns the child pid to the parent process and `nil` to the child process. When called with the block, it creates the child process and execute the block in the child process.

`gets([rs])`
`readline([rs])`

Reads a string from the virtual concatenation of each file listed on the command line or standard input (in case no files specified). If the end of file is reached, `nil` will be the result. The line read is also set to the variable `$_`. The line terminator is specified by the optional argument *rs*, which default value is defined by the variable `$/`.

`readline` functions just like `gets`, except it raises an `EOFError` exception at the end of file.

`global_variables`

Returns the list of the global variable names defined in the program.

`gsub(pattern[, replace])`
`gsub!(pattern[, replace])`

Searches a string held in the variable `$_` for a *pattern*, and if found, replaces all the occurrence of the pattern with the *replace* and returns the replaced string. `gsub!` modifies the original string in place, `gsub` makes copy, and keeps the original unchanged. See also String#gsub.

`iterator?`

Returns true, if called from within the methods called with the block (the iterators), otherwise false.

`load(file[, priv])`

Loads and evaluates the Ruby program in the *file*. If *file* is not an absolute path, it searches file to be load from the search path in the variable `$:`. The tilde (`~') at begenning of the path will be expanded into the user's home directory like some shells.

If the optional argument *priv* is true, loading and evaluating is done under the unnamed module, to avoid global name space pollution.

`local_variables`

Returns the list of the local variable names defined in the current scope.

`loop`

Loops forever (until terminated explicitly).

`open(`*`file`*`[, `*`mode`*`])`
`open(`*`file`*`[, `*`mode`*`]){...}`

Opens the *file*, and returns a File object associated with the file. The *mode* argument specifies the mode for the opened file, which is either `"r"`, `"r+"`, `"w"`, `"w+"`, `"a"`, `"a+"`. See `fopen(3)`. If *mode* omitted, the default is `"r"`

If the *file* begins with `"|"`, Ruby performs following string as a sub-process, and associates pipes to the standard input/output of the sub-process.

Note for the converts from Perl: The command string **starts** with `` `| ``', not ends with `` `| ``'.

If the command name described above is `"-"`, Ruby forks, and create pipe-line to the child process.

When `open` is called with the block, it opens the file and evaluates the block, then after the evaluation, the file is closed for sure. That is:

```
open(path, mode) do |f|
  ...
end

# mostly same as above

f = open(path, mode)
begin
  ...
ensure
  f.close
end
```

`p(`*`obj`*`)`

Prints human-readable representation of the *obj* to the stdout. It works just like:

```
print obj.inspect, "\n"
```

`print(`*`arg`*`1...)`

Prints arguments. If no argument given, the value of the variable `$_` will be printed. If an argument is not a string, it is converted into string using `Kernel#to_s`.

If the value of `$;` is non-nil, its value printed between each argument. If the value of `$\` is non-nil, its value printed at the end.

`printf([`*`port`*`, ]`*`format`*`, `*`arg`*`...)`

Prints arguments formatted according to the *format* like `sprintf`. If the first argument is the instance of the IO or its subclass, print redirected to that object. the default is the value of `$stdout`.

`proc`
`lambda`

Returns newly created procedure object from the block. The procedure object is the instance of the class Proc.

`putc(`*`c`*`)`

Writes the character *c* to the default output ($>).

`putc(`*`obj`*`..)`

Writes an *obj* to the default output ($>), then newline for each arguments.

`raise([`*`error_type`*`,][`*`message`*`][,`*`traceback`*`])`
`fail([`*`error_type`*`,][`*`message`*`][,`*`traceback`*`])`

Raises an exception. In no argument given, re-raises last exception. With one arguments, raises the exception if the argument is the exception. If the argument is the string, `raise` creates a new `RuntimeError` exception, and raises it. If two arguments supplied, `raise` creates a new exception of type *error_type*, and raises it.

If the optional third argument *traceback* is specified, it must be the traceback infomation for the raising exception in the format given by variable *$@* or `caller` function.

The exception is assigned to the variable `$!`, and the position in the source file is assigned to the `$@`.

If the first argument is not an exception class or object, the exception actually raised is determined by calling it's `exception` method (baring the case when the argument is a string in the second form). The `exception` method of that class or object must return it's representation as an exception.

The `fail` is an alias of the `raise`.

`rand(`*`max`*`)`

Returns a random integer number greater than or equal to 0 and less than the value of *max*. (*max* should be positive.) Automatically calls `srand` unless `srand()` has already been called.

If *max* is 0, `rand` returns a random float number greater than or equal to 0 and less than 1.

`readlines([`*`rs`*`])`

Reads entire lines from the virtual concatenation of each file listed on the command line or standard input (in case no files specified), and returns an array containing the lines read.

Lines are separated by the value of the optional argument *rs*, which default value is defined by the variable *$/*.

`require(`*`feature`*`)`

Demands a library file specified by the *feature*. The *feature* is a string to specify the module to load. If the extension in the *feature* is `".so"`, then Ruby interpreter tries to load dynamic-load file. If the extension is `".rb"`, then Ruby script will be loaded. If no extension present, the interpreter searches for dynamic-load modules first, then tries to Ruby script. On some system actual dynamic-load modules have extension name `".o"`, `".dll"` or something, though `require` always uses the extension `".so"` as a dynamic-load modules.

`require` returns true if modules actually loaded. Loaded module names are appended in `$"`.

`select(`*`reads`*`[, `*`writes`*`[, `*`excepts`*`[, `*`timeout`*`]]])`

Calls `select(2)` system call. *Reads*, *writes*, *excepts* are specified arrays containing instances of the IO class (or its subclass), or `nil`.

The *timeout* must be either an integer, `Float`, `Time`, or `nil`. If the *timeout* is `nil`, `select` would not time out.

`select` returns `nil` in case of timeout, otherwise returns an array of 3 elements, which are subset of

argument arrays.

`sleep([sec])`

Causes the script to sleep for *sec* seconds, or forever if no argument given. May be interrupted by sending the process a `SIGALRM` or `run` from other threads (if thread available). Returns the number of seconds actually slept. *sec* may be a floating-point number.

`split([sep[, limit]])`

Return an array containing the fields of the string, using the string *sep* as a separator. The maximum number of the fields can be specified by *limit*.

`format(format...)`
`sprintf(format...)`

Returns a string formatted according to a *format* like usual printf conventions of the C language. See `sprintf(3)` or `printf(3)`. In addition, `sprintf` accepts `%b` for binary. Ruby does not have unsigned integers, so unsigned specifier, such as %b, %o, or %x, converts negative integers into 2's complement form like `%..f`. supplying sign (`+`, `-`) or space option for the unsigned specifier changes its behavior to convert them in absolute value following `-` sign.

`srand([seed])`

Sets the random number seed for the `rand`. If *seed* is omitted, uses the current time etc. as a seed.

`sub(pattern[, replace])`
`sub!(pattern[, replace])`

Searches a string held in the variable `$_` for a *pattern*, and if found, replaces the first occurrence of the pattern with the *replace* and returns the replaced string. `sub!` modifies the original string in place, `sub` makes copy, and keeps the original unchanged. See also String#sub.

`syscall(num, arg...)`

Calls the system call specified as the first arguments, passing remaining as arguments to the system call. The arguments must be either a string or an integer.

`system(command...)`

Perform *command* in the sub-process, wait for the sub-process to terminate, then return true if it successfully exits, otherwise false. Actual exit status of the sub-process can be found in `$?`.

If multiple arguments are given, `system` invokes command directly, so that whitespaces and shell's meta-characters are not processed by the shell.

See `exec` for the execution detail.

`test(cmd, file [, file])`

Does a file test. the *cmd* would be one of following:

- commands which takes one operand:

`?r`	File is readable by effective uid/gid.
`?w`	File is writable by effective uid/gid.
`?x`	File is executable by effective uid/gid.
`?o`	File is owned by effective uid.
`?R`	File is readable by real uid/gid.
`?W`	File is writable by real uid/gid.

`?X`	File is executable by real uid/gid.
`?O`	File is owned by real uid.
`?e`	File exists.
`?z`	File has zero size.
`?s`	File has non-zero size (returns size).
`?f`	File is a plain file.
`?d`	File is a directory.
`?l`	File is a symbolic link.
`?p`	File is a named pipe (FIFO).
`?S`	File is a socket.
`?b`	File is a block special file.
`?c`	File is a character special file.
`?u`	File has setuid bit set.
`?g`	File has setgid bit set.
`?k`	File has sticky bit set.
`?M`	File last modify time.
`?A`	File last access time
`?C`	File last status change time.

- commands which takes two operands:

`?=`	Both files have same modify time.
`?>`	File1 is newer than file2.
`?<`	File1 is older than file2.
`?-`	File1 is a hard link to file2

`throw(`*`tag`*`[, `*`value`*`])`

Casts an non-local exit to the enclosing `catch` waiting for *tag*, or terminates the program if no such `catch` waiting. The *tag* must be the name of the non-local exit, which is either a symbol or a string. `catch` may not appear in the same method body. the *value* will be the return value of the `catch`. The default value is the `nil`.

`trace_var(`*`variable`*`, `*`command`*`)`
`trace_var(`*`variable`*`) {...}`

Sets the hook to the *variable*, which is called when the value of the variable changed. the *variable* must be specified by the symbol. the *command* is either a string or a procedure object. To remove hooks, specify `nil` as a *command* or use `untrace_var`.

`trap(`*`signal`*`, `*`command`*`)`
`trap(`*`signal`*`) {...}`

Specifies the signal handler for the *signal*. The handler *command* must be either a string or a procedure object. If the *command* is a string `"SIG_IGN"` or `"IGNORE"`, then specified signal will be ignored (if possible). If the *command* is a string `"SIG_DFL"` or `"DEFAULT"`, then system's default action will be took for the signal.

The special signal 0 or `"EXIT"` is for the termination of the script. The signal handler for `EXIT` will be called just before the interpreter terminates.

`untrace_var(`*`variable`*`[, `*`command`*`])`

Deletes the hook associated with the *variable*. If the second argument omitted, all the hooks will be removed. `trace_var` returns an array containing removed hooks.

Pre-defined variables

- $!
- $@
- $&
- $`
- $'
- $+
- $1, $2..
- $~
- $=
- $/
- $\
- $,
- $;
- $.
- $<
- $>
- $_
- $0
- $*
- $$
- $?
- $:
- $"
- $DEBUG
- $FILENAME
- $LOAD_PATH
- $stdin
- $stdout
- $stderr
- $VERBOSE
- option variables
    - $-0
    - $-a
    - $-d
    - $-F
    - $-i
    - $-I
    - $-l
    - $-p
    - $-v

$!

The exception information message. `raise` sets this variable.

$@

The backtrace of the last exception, which is the array of the string that indicates the point where methods invoked from. The elements in the format like:

```
"filename:line"
```

or

```
"filename:line:in `methodname'"
```

(Mnemonic: where exception occurred **at**.)

`$&`

The string matched by the last successful pattern match in this scope, or `nil` if the last pattern match failed. (Mnemonic: like & in some editors.) This variable is read-only.

`$``

The string preceding whatever was matched by the last successful pattern match in the current scope, or `nil` if the last pattern match failed. (Mnemonic: ` often precedes a quoted string.) This variable is read-only.

`$'`

The string following whatever was matched by the last successful pattern match in the current scope, or `nil` if the last pattern match failed. (Mnemonic: ' often follows a quoted string.)

`$+`

The last bracket matched by the last successful search pattern, or `nil` if the last pattern match failed. This is useful if you don't know which of a set of alternative patterns matched. (Mnemonic: be positive and forward looking.)

`$1, $2...`

Contains the subpattern from the corresponding set of parentheses in the last successful pattern matched, not counting patterns matched in nested blocks that have been exited already, or `nil` if the last pattern match failed. (Mnemonic: like \digit.) These variables are all read-only.

`$~`

The information about the last match in the current scope. Setting this variables affects the match variables like `$&`, `$+`, `$1`, `$2`.. etc. The nth subexpression can be retrieved by `$~[nth]`. (Mnemonic: ~ is for match.) This variable is locally scoped.

`$=`

The flag for case insensitive, `nil` by default. (Mnemonic: = is for comparison.)

`$/`

The input record separator, newline by default. Works like `awk`'s RS variable. If it is set to `nil`, whole file will be read at once. (Mnemonic: / is used to delimit line boundaries when quoting poetry.)

`$\`

The output record separator for the `print` and `IO#write`. The default is `nil`. (Mnemonic: It's just like /, but it's what you get "back" from Ruby.)

`$,`

The output field separator for the `print`. Also, it is the default separator for `Array#join`. (Mnemonic: what is printed when there is a , in your print statement.)

`$;`

The default separator for `String#split`.

`$.`

The current input line number of the last file that was read.

`$<`

The virtual concatenation file of the files given by command line arguments, or stdin (in case no argument file supplied). `$<.file` returns the current filename. (Mnemonic: $< is a shell input source.)

`$>`

The default output for `print`, `printf`. `$stdout` by default. (Mnemonic: $> is for shell output.)

`$_`

The last input line of string by `gets` or `readline`. It is set to `nil` if `gets`/`readline` meet `EOF`. This variable is locally scoped. (Mnemonic: partly same as Perl.)

`$0`

Contains the name of the file containing the Ruby script being executed. On some operating systems assigning to `$0` modifies the argument area that the `ps(1)` program sees. This is more useful as a way of indicating the current program state than it is for hiding the program you're running. (Mnemonic: same as sh and ksh.)

`$*`

Command line arguments given for the script. The options for Ruby interpreter are already removed. (Mnemonic: same as sh and ksh.)

`$$`

The process number of the Ruby running this script. (Mnemonic: same as shells.)

`$?`

The status of the last executed child process.

`$:`

The array contains the list of places to look for Ruby scripts and binary modules by `load` or `require`. It initially consists of the arguments to any `-I` command line switches, followed by the default Ruby library, probabl "/usr/local/lib/ruby", followed by ".", to represent the current directory. (Mnemonic: colon is the separators for PATH environment variable.)

`$"`

The array contains the module names loaded by `require`. Used for prevent `require` from load modules twice. (Mnemonic: prevent files to be doubly quoted(loaded).)

`$DEBUG`

The status of the `-d` switch.

`$FILENAME`

Same as `$<.filename`.

`$LOAD_PATH`

The alias to the `$:`.

`$stdin`

The current standard input.

`$stdout`

The current standard output.

`$stderr`

The current standard error output.

`$VERBOSE`

The verbose flag, which is set by the `-v` switch to the Ruby interpreter.

option variables

The variables which names are in the form of `$-?`, where ? is the option character, are called option variables and contains the information about interpreter command line options.

`$-0`

The alias to the `$/`.

`$-a`

True if option `-a` is set. Read-only variable.

`$-d`

The alias to the `$DEBUG`.

`$-F`

The alias to the `$;`.

`$-i`

In in-place-edit mode, this variable holds the extention, otherwise `nil`. Can be assigned to enable (or disable) in-place-edit mode.

`$-I`

The alias to the `$:`.

`$-l`

True if option `-l`is set. Read-only variable.

`$-p`

True if option `-p` is set. Read-only variable.

`$-v`

The alias to the `$VERBOSE`.

Pre-defined global constants

- `TRUE`
- `FALSE`
- `NIL`
- `STDIN`
- `STDOUT`
- `STDERR`
- `ENV`
- `ARGF`
- `ARGV`
- `DATA`
- `RUBY_VERSION`
- `RUBY_RELEASE_DATE`
- `RUBY_PLATFORM`

`TRUE`

The typcal true value. All non-false values (everything except `nil` and `false`) is true in Ruby.

`FALSE`

The false itself.

`NIL`

The nil itself.

`STDIN`

The standard input. The default value for `$stdin`.

`STDOUT`

The standard output. The default value for `$stdout`.

`STDERR`

The standard error output. The default value for `$stderr`.

`ENV`

The hash-like object contains current environment variables. Setting a value in `ENV` changes the environment for child processes.

`ARGF`

The alias to the `$<`.

`ARGV`

The alias to the `$*`.

`DATA`

The file object of the script, pointing just after the `__END__`. Not defined unless the script is not read from the file.

`VERSION`

The Ruby version string.

`RUBY_RELEASE_DATE`

The relase date string.

`RUBY_PLATFORM`

The platform identifier.

ENV

The object that represents environment variables. It has almost equivalent interfaces to the Hash objects, except the keys and values of ENV must be strings.

Included Modules:

`Enumerable`

Methods:

`self [`*key*`]`

Returns the value of the environment variable specified by the *key*. Returns `nil`, if no such environment variable exists.

`self [`*key*`]= `*value*

Changes or sets the value of the environment variable specified by the *key* to the *value*. If the *value* is `nil`, the environment variable will be removed.

`delete(`*key*`)`

Deletes the environment variable. Returns nil if the variable specified by the *key* does not exist. In case the block is supplied to this method, it will be evaluated when no environment variable exists.

```
delete_if {|key, value|...}
reject!{|key, value|...}
```

Deletes environment variables, if the evaluated value of the block with the array of `[key,value]` as an argument is true.

```
each {|key, value|...}
each_pair {|key, value|...}
```

Iterates over each pair of keys and values (`[key,value]`) of the environment variables.

```
each_key {|key|...}
```

Iterates over each name of all environment variables.

```
each_value {|value|...}
```

Iterates over each value of all environment variables.

```
empty?
```

Returns true if no environment variable defined.

```
has_key?(val)
key?(val)
include?(val)
```

Returns true if there is the environment variable named *val*.

```
has_value?(value)
value?(value)
```

Returns true, if there exists any environment variable which has the *value*.

```
indexes(key_1,..., key_n)
```

Returns an array of values of environment variables that names are specified by arguments.

```
keys
```

Returns an array of all environment variables.

```
length
size
```

Returns the number of environment variables.

```
to_a
```

Returns an array of two element arrays of the environment variables which is `[name,value]`.

```
values
```

Returns an array of the values of all environment variables.

Pre-defined classes and modules

Pre-defined classes

- Object
  - Array
  - Data
  - Dir
  - Exception
    - Interrupt
    - NotImplementError
    - SignalException
    - StandardError
      - ArgumentError
      - FloatDomainError
      - IOError
        - EOFError
      - IndexError
      - LoadError
      - LocalJumpError
      - NameError
      - RuntimeError
      - SecurityError
      - SyntaxError
      - SystemCallError
        - Errno::E*
      - SystemStackError
      - ThreadError
      - TypeError
      - ZeroDivisionError
    - SystemExit
    - fatal
  - Hash
  - IO
    - File
  - MatchingData
  - Module
    - Class
  - Numeric
    - Integer
      - Bignum
      - Fixnum
    - Float
  - Proc
  - Range
  - Regexp
  - String
  - Struct
  - Time
  - NilClass

Pre-defined modules

- Comparable
- Enumerable
- Errno
- FileTest
- GC

- Kernel
- Marshal
- Math
- ObjectSpace
- Precision
- Process

Object

The super class of all the Ruby class hierarchy.

SuperClass:

Kernel

Methods:

`self == other`

Checks if two objects are **equal**. The default definition in the `Kernel` class checks by object ID. It should be redefined in subclasses according to the characteristic of the class.

`clone`
`dup`

Returns the copy of the object. For cloned object,

```
obj == obj.clone
```

is always true, though

```
obj.equal?(obj.clone)
```

is false most of the time.

the `dup` method is defined as:

```
def dup
  self.clone
end
```

`clone` and `dup` copy the receiver, but they do not copy the objects pointed by the receiver (i.e shallow copy).

`display([port])`

Prints `self` to the *port*. The *port*'s default value is `$>`.

```
def display(out=$>)
  out.print self
end
```

```
public :display
```

`eql?(`*other*`)`

Checks if two objects are **equal**. This method is used by `Hash` to compare whether two keys are same. When this method is redefined, the `hash` method should be updated.

The default definition of the method `eql?` is like blow:

```
def eql?(other)
  self == other
end
```

`equal?(`*other*`)`

Checks if two objects have same object ID. This method should not be redefined in the subclass.

`self =~ `*other*

Method for old style match like `obj =~ /RE/`. The default definitions is `==`'.

`self === `*other*

This method is used to compare in `case`. The default definitions is `==`'.

`extend(module...)`

Extends `self` by adding methods defined in *module*s as singleton-methods.

`hash`

Returns the integer hash value of the object. It is used in the `Hash` class to calculate the holding place of an object. The hash value of two objects must be equal, where these two objects are equal by using `eql?` operator. So, whenever you redefine the definition of the `eql?` operator in any class, don't forget to redefine `hash` method according to the definition.

`id`

Returns the unique integer value for each object.

`inspect`

Returns the human-readable string representation of the receiver.

`initialize(...)`

The initialize method for the user-defined classes. This method will be called from `Class#new` to initialize the newly created object. The default behavior is to do nothing. It is assumed that this method will be redefined in the subclasses. The argument to `Class#new` will be passed to the `initialize`

the method named `initialize` automatically falls under private visibility.

`instance_eval(`*expr*`)`
`instance_eval{...}`

Evaluates the *expr* string in the object context. If block supplied for the method, the block is evaluated under the receiver's context. This allows to avoid compiling the string everytime.

In *instance_eval*'s context, `self` is bound to the object, so that instance variables and methods of the receiver are accessed.

`instance_of?(`*`class`*`)`

Returns true, if `self` is an instance of the specified *class*. It is always true, when `obj.kind_of?(c)` is true.

`instance_variables()`

Returns the array of `self`'s instance variable names.

`kind_of?(`*`class`*`)`
`is_a?(`*`class`*`)`

Returns true, if `self` is an instance of the specified *class*, or its subclass.

`method_missing(`*`msg_id`*`, ...)`

Will be called when the specified method is not defined for the receiver. The first argument, *msg_id* is the method name as a symbol. The second and later arguments are the arguments given to the undefined method, if any.

`methods`

Returns the array of the public method names which defined in the receiver.

`nil?`

Checks whether the receiver is `nil` or not.

`private_methods`

Returns the array of the private method names which defined in the receiver.

`remove_instance_variable(`*`name`*`)`

Removes specified instance variable from the object. It does not raise any exception even if the object does not have named instance variable.

`respond_to?(`*`mesg`*`[,`*`priv`*`])`

Returns true if the receiver has the public method named *mesg*. *Mesg* must be the symbol fixnum or the string. It returns true for private methods, if the optional argument *priv* given, and its value is true,

`send(`*`symbol`*`[, `*`args`*`...])`

calls the method specified by the *symbol*, with *args*.

`singleton_method_added(`*`id`*`)`

Will be called when the singleton method defined for the receiver.

`singleton_methods`

Returns the array of the singleton method names which defined in the receiver.

`taint`

Turns on `taint mark' of the object.

`tainted?`

Returns true if the string is `tainted'.

`to_s`

Returns the string representation of the `self`. This method is used internally by `print` and `sprintf`.

`to_a`

Converts the `self` into an array. Returns an array of 1 element contains `self` for classes would not be converted into an array naturally.

`type`

Returns the receiver's class.

`untaint`

Turns off `taint mark' of the object. Programmers should be responsible for the security issue caused by removing the taint mark. The mark can't be removed if the security level ($SAFE) is greater than or equals to level 3.

Data

The wrapper class for the C pointer. Used mostly by the extension libraries.

SuperClass:

`Object`

Dir

The class for the directory stream.

SuperClass:

Object

Included Modules:

Enumerable

Class Methods:

self[*pattern*]
glob(*pattern*)

Expands a wild-card *pattern*, then returns the results in an array of strings.

Wild-cards are:

*	Matches any string, including the null string
?	Matches any single character
[ ]	Matches any one of enclosed character. A pair of characters separated by a `-' sign denotes a range. If the first character following `[' is a `^', then any character not enclosed is matched.
{ }	Expanded to combinations of comma separating string within braces. For example, `foo{a,b,c}' will be expanded into three strings, `fooa', `foob', `fooc'. The generated names need not to be exist.

chdir(*path*)

Changes the current directory to the *path*.

chroot(*path*)

Changes the root directory to *path*. See chroot(2). Only the super-user may change the root directory

There is no way to back up the old root directory.

getwd
pwd

Returns the absolute pathname of the current directory.

foreach(*path*)

iterates over the items in the directory specified by path. It works like:

```
dir = Dir.open(path)
begin
  dir.each {
    ...
  }
ensure
  dir.close
end
```

mkdir(*path*[, *mode*])

Creates a directory named *path*. the permission of a newly created directory specified by *mode*. The default mode is `0777`, which will be modified by users `umask (mode & ~umask)`.

`open(`*`path`*`)`

Opens the directory stream corresponding to the directory *path*.

`delete(`*`path`*`)`
`rmdir(`*`path`*`)`
`unlink(`*`path`*`)`

Deletes the directory, which must be empty.

Methods:

`close`

Closes the directory stream. Later operation to the directory stream raises an exception.

`each {|`*`item`*`|...}`

Gives each item in the directory as the block parameter.

`read`

Returns the next item in the directory stream.

`rewind`

Resets the position of the directory stream to the beginning of the directory.

`seek(`*`pos`*`)`

Sets the location of the directory stream to the *pos*. The *pos* must be an offset returned by `Dir#tell`.

`tell`

Returns the current position of the directory stream.

`---`

Exception

The ancester of the all exceptions.

SuperClass:

`Object`

Class Methods:

```
exception([error_message = ""])
new([error_message = ""])
```

Creates a new exception object. The error_message string can be supplied as the optional argument. The default exception handler at top leve, will show this error_message.

`exception` is used internally by `raise`.

Methods:

`backtrace`

Returns backtrace information as the array of the strings in the following format.

```
"#{sourcefile}:#{sourceline}:in `#{method}'"
      (within methods)
"#{sourcefile}:#{sourceline}"
      (at top level)
```

`exception`

Return self. This method is used internally by `raise`.

`message`
`to_s`
`to_str`

Returns the error message string.

`set_backtrace(errinfo)`

Sets the backtrace information of the exception. *errinfo* must be the array of the strings.

Interrupt

The exception for the untrapped `SIGINT`.

SuperClass:

`Exception`

NotImplementError

The exception raised when unimplemented feature was invoked.

SuperClass:

`Exception`

SignalException

The exception raised by the signals (except SIGINT).

SuperClass:

`Exception`

StandardError

The `rescue` class without class specifified will catch the subclasses of this exception class.

SuperClass:

`Exception`

SystemExit

Raised by `exit` to terminate the interpreter.

SuperClass:

`Exception`

---

fatal

The fatal error, which can not be caught nor handled.
e.g.:

- No more memory available
- Thread dead-lock happened
- Could not move to the directory specified by options *-x* and *-X*.
- Could not perform inplace edit by some reasons

SuperClass:

`Exception`

ArgumentError

The exception raised when some problem found with methods arguments. E.g.

```
Math.sqrt(-1)
```

SuperClass:

`StandardError`

FloatDomainError

Raised by the floating point number operation with infinite number or NaN.

SuperClass:

`StandardError`

IndexError

Raised when index out of range.

SuperClass:

`StandardError`

IOError

Raised when I/O error happened.

SuperClass:

`StandardError`

LoadError

Raised when `load` or `require` failed.

SuperClass:

`StandardError`

LocalJumpError

raised by local jumps (`return`, `break`, `next`, `redo`, `retry`) from a `Proc` object which is brought out of its original scope.

SuperClass:

`StandardError`

NameError

Raised by accessing an undefined identifier.

SuperClass:

`StandardError`

RuntimeError

The default exception caused by `raise` without specifying an exception.

SuperClass:

`StandardError`

SecurityError

Raised by security problem with tainted data.

SuperClass:

`StandardError`

SyntaxError

Raised by a syntax error.

SuperClass:

`StandardError`

SystemCallError

The exceptions for system call failures.
This is the abstract super class for the actual system call exception classes, which are defined under the `Errno` module.

SuperClass:

`StandardError`

Methods:

`errno`

Returns the `errno` passed from the OS.

SystemStackError

Raised when stack level becomes too deep.

SuperClass:

`StandardError`

TypeError

Raised by the type problem. Mostly caused by C extensions.

SuperClass:

`StandardError`

ThreadError

Raised by the follwing `Thread`errors:

- try to `join` the current thread.
- `join` causes dead lock.
- try to `wakeup` the dead thread.
- try to `stop` a sole thread.
- try to create a thread without the block.
- global jump out of thread by `throw` happened.
- exit from the thread block by `return`.

SuperClass:

`StandardError`

ZeroDivisionError

Raised by a division by zero.

SuperClass:

`StandardError`

EOFError

Raised when EOF is reached.

SuperClass:

IOError

Hash

The class for associative arrays or hash tables. The hash table can contain the association from any kind of object to any kind of object. The hash table allocation can be done by the hash expression:

```
{a=>b,...}
```

The hash value for the key value is calculated using method `Kernel#hash`. Since the hash value of the key should not be changed, instances of the classes like `Array`, `Hash`, etc. are not suited for the key. When the `string` used as the key, hash table copies and freeze it and use copied string as the key string. Attempt to Modify the freezed key strings raises an exception.

SuperClass:

Object

Included Modules:

Enumerable

Class Methods:

`Hash[`*`key, value...`*`]`

Creates a new hash table, interpreting arguments as key - value pair. The number of arguments must be times of two.

`new([`*`ifnone`*`])`

Creates a new, empty hash table. *ifnone* is the default value for the non-registered key.

Methods:

`self [`*`key`*`]`

Returns the value associated to the *key*. Returns the default value (or `nil`), if `key` is not registered in the hash table.

`self [`*`key`*`]= `*`value`*

Adds binding of *key* to *value*. If the *value* is `nil`, the binding from *key* will be removed. That means the hash table cannot holds the association to `nil`.

`clear`

Makes the hash table empty.

`default`

Returns the default value for the hash table.

`default=(`*`value`*`)`

Sets the default value for the hash table.

`delete(`*`key`*`)`

Deletes the association from `key`. Returns `nil` if *key* is not found in the hash table.

If the block supplied to the method, it will be evaluated when no association matches to *key*.

`delete_if {|`*`key`*`, `*`value`*`|...}`
`reject!{|`*`key`*`, `*`value`*`|...}`

Deletes association from hash table, if the evaluated value of the block with the array of `[key,value]` as an argument is true.

`dup`

Returns a newly created hash table which has the save keys and values to the receiver. `clone` returns the complete copy of the original hash table, including freeze status and instance variables. On the other hand, `dup` copies the hash table containts.

`each {|`*`key`*`, `*`value`*`|...}`
`each_pair {|`*`key`*`, `*`value`*`|...}`

Iterates over each pair of keys and values (`[key,value]`) of the hash.

`each_key {|`*`key`*`|...}`

Iterates over each key in the hash table.

`each_value {|`*`value`*`|...}`

Iterates over each value in the hash table.

`empty?`

Returns true, if hash table is empty.

`fetch(`*`key`*`[,`*`default`*`])`

Returns the value associated to the *key*. Returns *default*, if *key* is not registered in the hash table. If a block is given, evaluates and returns its value on value absence.

`freeze`

Prohibits modification of the hash table. Modification to the freezed string raises an exception.

`frozen`

Returns true if the hash table is frozen.

`has_key?(`*`key`*`)`
`key?(`*`key`*`)`
`include?(`*`key`*`)`

Returns true if hash table has *key* in the associations.

`has_value?(`*`value`*`)`
`value?(`*`value`*`)`

Returns true if hash table has *value* in the associations.

`index(`*`val`*`)`

Returns the key corresponding to *val*. Returns `nil` if there is no matching value.

`indexes(`*`key_1`*`,..., `*`key_n`*`)`
`indices(`*`key_1`*`,..., `*`key_n`*`)`

Returns an array of values which keys are specified by arguments.

`keys`

Returns an array of keys in the hash table.

`length`
`size`

Returns the number of associations in the hash table.

`invert`

Returns the value-to-key mapping of the hash.

`replace(`*`other`*`)`

Copis the content of *other* into the hash.

`shift`

Removes and returns an association from the hash table. The association is in the form `[key,value]`.

`store(`*`key`*`,`*`value`*`)`

Adds binding of *key* to *value*.

`to_a`

Returns an array of two element arrays of associations which is `[key,value]`.

`update(`*other*`)`

Merges the contents of another hash, overriding existing keys.

`values`

Returns an array of the values in the hash table.

IO

The `IO` class provides the basic IO feature.

SuperClass:

`Object`

Included Modules:

`Enumerable`

Class Methods:

`foreach(`*path*`[, `*rs*`])`

Iterates over each line from the IO port specified by `path`. It works like:

```
port = open(path)
begin
  port.each_line {
    ...
  }
ensure
  port.close
end
```

Lines are separated by the value of the optional argument *rs*, which default value is defined by the variable `$/`.

`new(`*fd*`[,`*mode*`])`

Creates a stream associated with the file descriptor *fd*.

`popen(`*command* `[, `*mode*`])`

Performs the *command* as a sub-process, and associates pipes to the standard input/output of the sub-process. The *mode* argument specifies the mode for the opened IO port, which is either `"r"`, `"r+"`,

"w", "w+", "a", "a+". If *mode* omitted, the default is "r"

If the command name is "-", Ruby forks, and create pipe-line to the child process.

`pipe`

Opens a pair of connected pipes like the corresponding system call, and returns them in the array of two elements (read-side first, write-side next).

`readlines(`*`path`*`[, `*`rs`*`])`

Reads entire lines from the IO port specified by `path` and returns an array containing the lines read. It works like:

```
port = open(path)
begin
  port.each_line {
    ...
  }
ensure
  port.close
end
```

Lines are separated by the value of the optional argument *rs*, which default value is defined by the variable `$/`.

`select(`*`reads`*`[, `*`writes`*`[, `*`excepts`*`[, `*`timeout`*`]]])`

Calls `select(2)` system call. *Reads*, *writes*, *excepts* are specified arrays containing instances of the IO class (or its subclass), or `nil`.

The *timeout* must be either an integer, Float, `Time`, or `nil`. If the *timeout* is `nil`, `select` would not time out.

`select` returns `nil` in case of timeout, otherwise returns an array of 3 elements, which are subset of argument arrays.

Methods:

`self << `*`object`*

Output *object* to the IO port. If *object* is not a string, it will be converted into the string using `to_s`. This method returns self, so that the code below works:

```
$stdout << 1 << " is a " << Fixnum << "\n"
```

`binmode`

Changes the stream into binary mode. This is useful only under MSDOS. There's no way to reset to ascii mode except re-opening the stream.

`close`

Closes the IO port. All operations on the closed IO port will raise an exception. IO ports are automatically closed when they are garbage-collected.

`closed?`

Returns true if the IO port closed.

`each([rs]) {|line|...}`
`each_line([rs]) {|line|...}`

Iterates over each line from the IO port. The IO port must be opened in read-mode. (See `open`)

Lines are separated by the value of the optional argument *rs*, which default value is defined by the variable `$/`.

`each_byte {|ch|...}`

Reads byte by byte from the IO port. The IO port must be opened in read-mode. (See `open`)

`eof`
`eof?`

Returns true it the stream reaches end of file.

`fcntl(cmd, arg)`

Performs system call `fcntl` on the IO object. See `fcntl(2)` for detail.

If the *arg* is a number, the numeric value is passed to the system call. If the `arg` is a string, it is treated as the packed structure. The default *arg* value is 0.

`fileno`
`to_i`

Returns the file descriptor number of the IO port.

`flush`

Flushes the internal buffer of the IO port.

`getc`

Reads the next character from the IO port, and returns an fixnum corresponding that character. Returns `nil` at the end of file.

`gets([rs])`

Reads a line from the IO port, or `nil` on end of file. Works mostly same as `each`, but `gets` does not iterate.

Lines are separated by the value of the optional argument *rs*, which default value is defined by the variable `$/`.

The line read is set to the variable `$_`.

`ioctl(cmd, arg)`

Performs system call `ioctl` on the IO object. See `ioctl(2)` for detail.

If the *arg* is a number, the numeric value is passed to the system call. If the `arg` is a string, it is treated as the packed structure. The default *arg* value is 0.

`isatty`
`tty?`

Returns true if the IO port connected to the tty.

`lineno`

Returns the current line number of the IO.

`lineno= number`

Sets the line number of the IO.

`pos`

Returns the current position of the file pointer.

`pos= pos`

Moves the file pointer to the *pos*.

`print arg...`

Outputs arguments to the IO port.

`printf(format, arg...)`

Output arguments to the IO port with formatting like `printf` in C language.

`putc(c)`

Writes the character *c* to the stream.

`puts(obj...)`

Outputs an *obj* to the IO port, then newline for each arguments.

`read [length]`

Attempts to read *length* bytes of data from the IO port. If no *length* given, reads all data until `EOF`.

returns *nil* at `EOF`.

`readchar`

Reads a character from the IO port, just like `getc`, but raises an `EOFError` exception at the end of file.

`readline([rs])`

Reads a line from the IO port, just like `gets`, but raises an `EOFError` exception at the end of file.

Each lines is separated by the value of the optional argument *rs*, which default value is defined by the variable `$/`.

The line read is set to the variable `$_` just like `IO#gets`.

`readlines([rs])`

Reads entire lines from the IO port and returns an array containing the lines read.

Lines are separated by the value of the optional argument *rs*, which default value is defined by the variable `$/`.

`reopen(`*`io`*`)`

Reconnect `self` to *io*. It also changes the class of the stream.

`rewind`

Resets the position of the file pointer to the beginning of the file.

`seek(`*`offset, whence`*`)`

Moves the file pointer to the *offset*. The value for *whence* are 0 th set the file pointer to *offset*, 1 to set it to current plus *offset*, 2 to set it to EOF plus offset.

`stat`

Returns the status info of the file in the `Stat` structure, which has attributes as follows:

```
dev             # device number of file-system
ino             # i-node number
mode            # file mode
nlink           # number of hard links
uid             # user ID of owner
gid             # group ID of owner
rdev            # device type (special files only)
size            # total size, in bytes
blksize         # preferred blocksize for file-system I/O
blocks          # number of blocks allocated
atime           # time of last access
mtime           # time of last modification
ctime           # time of last i-node change
```

For more detail, see `stat(2)`. Some fields are filled with 0, if that field is not supported on your system.

`sync`

Returns the `sync' mode of the IO port. When the `sync' mode is true, the internal buffer will be flushed, everytime something written to the output port.

`sync= `*`newstate`*

Sets the `sync' mode of the IO port.

`sysread(`*`length`*`)`

Attempts to read *length* bytes of data from the IO port, using the system call read(2). It bypasses stdio, so mixing this with other kinds of reads/eof checks may cause confusion.

`syswrite(`*`string`*`)`

Attempts to write data from the *string* to the IO port, using the write(2) system call. It bypasses stdio, so mixing this with prints may cause confusion.

`tell`

Returns the current position of the file pointer.

`write(str)`

Outputs the string to the IO port. Returns the number of bytes written.

`ungetc(c)`

Pushes `c` back to the stream. Only one push-back is guaranteed.

File

The class for accessing files. Normally, created by `open`. This class has singleton-methods that the `FileTest` class has.

SuperClass:

`IO`

Class Methods:

`atime(filename)`

Returns the last accessed time of the file.

`basename(filename[, suffix])`

Returns the last slash-delimited component of the *filename*. If *suffix* is supplied and is identical to the end of name, it is removed from filename.

Example:

```
basename("ruby/ruby.c")
         => "ruby.c"
basename("ruby/ruby.c", ".c")
        => "ruby"
```

`ctime(filename)`

Returns the last status change time of the file.

`chmod(mode, path, file...)`

Change the mode of the `file`s to the *mode*. Returns the number of the file it processed. See `chmod(2)`.

`chown(owner, group, file...)`

Change the owner and the groups of the `file`s to the *owner*. Only super-user can change the owner of the file. Returns the number of the file it processed.

The owner and the group can leave unchanged with supplying `nil` or `-1` to as the argument.

`dirname(`*`filename`*`)`

Returns all but the final slash-delimited component of *filename*. Returns `"."` (meaning the current directory), if *filename* is a single component.

`expand_path(`*`path`*`[,`*`default_dir`*`])`

Converts *path* to absolute, and canonicalized path. Second arg *default_dir* is directory to start with if *path* is relative (does not start with slash); if *default_dir* is nil or missing, the current directory of the process is used. An initial `~' expands to your home directory. An initial `~USER' expands to USER's home directory.

```
expand_path("..")
       => "/home/matz/work"
expand_path("~")
       => "/home/matz"
expand_path("~matz")
       => "/home/matz"
```

`ftype(`*`filename`*`)`

Returns a string which describes the type of the file. The string is either:

```
"file"
"directory"
"characterSpecial"
"blockSpecial"
"fifo"
"link"
"socket"
```

`join(`*`item...`*`)`

Combines file names using `File::Separator`.

It works just like:

```
[items,..].join(File::Separator)
```

`link(`*`old, new`*`)`

Creates a new hard link to old (existing) file. See `link(2)`.

`lstat(`*`filename`*`)`

Does the same thing as the `stat`, but stats a symbolic link instead of the file the symbolic link points to.

`mtime(`*`filename`*`)`

Returns the last modified time of the file.

`open(`*`path`*`[, `*`mode`*`])`

`new(path[, mode])`

Opens the file specified by *path*, and returns a `File` object associated with that file.

The *mode* argument specifies the mode for the opened file, which is either `"r"`, `"r+"`, `"w"`, `"w+"`, `"a"`, `"a+"`. See `fopen(3)`. If *mode* omitted, the default is `"r"`.

if *mode* is a integer, it is considered like the second argument to `open(2)`. One of the flags must be either `RDONLY`, `WRONLY` or `RDWR`. Flags may also be bitwise-or'd with one or more of `APPEND`, `CREAT`, `EXCL`, `NONBLOCK`, `TRUNC`, `NOCTTY`, `BINARY`.

If a new file is created as part of opening it, permissions (an integer) is used to set the permissions for the new file in conjunction with the process's file mode creation mask. Permissions defaults to 0666.

If `open` is called with block, the block will be executed with a newly opened file object. The file object will be closed automatically after executing the block.

`readlink(path)`

Returns the symbolic link path as a string.

`rename(from, to)`

Renames file, moving it across the directories if required. See `rename(2)`. If *to*-file already exists, it will be overwritten.

`size(pathname)`

Returns the size of the file specified by the *pathname* in bytes.

`split(pathname)`

Splits the *pathname* in a pair `[head, tail]`, where *tail* is the last pathname component and *head* is everything leading up to that.

`stat(filename)`

Returns the status info of the *filename* in the `Stat` structure, which has attributes as follows:

```
dev             # device number of file-system
ino             # i-node number
mode            # file mode
nlink           # number of hard links
uid             # user ID of owner
gid             # group ID of owner
rdev            # device type (special files only)
size            # total size, in bytes
blksize         # preferred blocksize for file-system I/O
blocks          # number of blocks allocated
atime           # time of last access
mtime           # time of last modification
ctime           # time of last i-node change
```

For more detail, see `stat(2)`. Some fields are filled with 0, if that field is not supported on your system.

`symlink(old, new)`

Created a *new* symbolic link to the *old*-file.

`truncate(path, length)`

truncate the file specified by *path* to at most *length* byte.

`unlink(`*`file`*`...)`
`delete(`*`file`*`...)`

Deletes *file*s. Returns the number of files successfully deleted. Use `Dir.rmdir` instead to delete directories.

`umask([`*`umask`*`])`

Changes umask of the user. Returns the current mask. If *umask* is not specified, just current returns the current mask value.

`utime(`*`atime, mtime, file`*`...)`

Change the access and modification times on each *file*s. Returns the number of files successfully changed. The first two arguments must be either the number or the instance of the `Time` class.

In addition, the `File` class has class methods defined in `FileTest`.

Methods:

`atime`

Returns the last accessed time of the file.

`ctime`

Returns the last status change time of the file.

`chmod(`*`mode`*`)`

Change the mode of the `file`s to the *mode*. See `chmod(2)`.

`chown(`*`owner, group`*`)`

Change the owner and the groups of the `file`s to the *owner*. Only super-user can change the owner of the file. The owner and the group can be unchanched with supplying `nil` or `-1` to as the argument.

`eof`
`eof?`

Returns true, if the file pointer reached at the end of file.

`flock(`*`operation`*`)`

Applys or removes an advisory lock on the file. `flock()` returns false if `LOCK_NB` supplied as the operation and the call would block. Valid operations are given below:

`LOCK_SH`
: Shared lock. More than one process may hold a shared lock for a given file at a given time.

`LOCK_EX`
: Exclusive lock. Only one process may hold an exclusive lock for a given file at a given time.

`LOCK_UN`
: Unlock.

`LOCK_NB`

Don't block when locking. May be specified (by or'ing) along with one of the other operations.

These constants are defined under the class `File`.

`lstat`

Does the same thing as the `stat`, but stats a symbolic link instead of the file the symbolic link points to.

`mtime`

Returns the last modified time of the file.

`reopen(`*`io`*`)`

Reconnect `self` to *io*. It also changes the class of the stream.

`reopen(`*`path, mode`*`)`

Opens and reconnects the file specified by *path*, with *mode*.

`path`

Returns pathname of the opened file.

`stat`

Returns the status info of the file in the `Stat` structure.

`truncate(`*`length`*`)`

truncate the file to at most *length* byte. The file must be opened in the write mode.

Constants:

`Separator`

The separating character of the file path, which is normally `"/"`.

MatchingData

The status of the regular expression match. The value of the variable `$~` is the instance of this class.

SuperClass:

`Object`

Methods:

`size`
`length`

Returns the number of the subpatterns (i.e. $~.to_a.size).

`offset(`*n*`)`

Returns the offsets ([start, end]) of the nth subpattern. 0 means the whole match.

`begin(`*n*`)`

Returns the starting offset of the nth match. 0 means the whole match.

`end(`*n*`)`

Returns the end offset of the nth match. 0 means the whole match.

`to_a`

Returns the array of the subpatterns, which is [$&, $1, $2,...].

`self[`*n*`]`

Returns the nth subpattern.

`pre_match`

Returns the string preceding the match (i.e. $`).

`post_match`

Returns the string following the match (i.e. $').

`string`

Returns the string where the match was performed.

Module

The class of the modules.

SuperClass:

`Object`

Methods:

`self < other`
`self <= other`
`self > other`
`self >= other`

Comparison operators. `self` > *other* returns `true`, if `self` is the superclass or included modules of the *other*.

`self === obj`

Returns `true`, if *obj* is the instance of the `self`, or its descendants. That means `case` can be used for type check for for classes or modules.

`alias_method(new,old)`

Gives alias to methods. Differences between this method and `alias` are:

- specifies method by String or ID (Integer).
- no global variable aliasing by this method.

`append_features(module_or_class)`

Append features (mothods and constants) to the receiver. `Module#include` is defined using this method.

`attr(name[, assignable])`

Defines new attribute and its access method to read, which are named `*name*' to the module. The access method definition is like this:

```
def attr; @attr; end
```

The optional second argument *assignable* is given, and its value is true, then the write method to the attribute is also defined. The write access method definition is like this:

```
def attr=(val); @attr = val; end
```

By re-defining the access method, accessing attribute can be altered. For example, defining the write access method like below, the assigned value can be printed.

```
attr("test", true)
def test=(val)
  print("test was ", @test, "\n")
  print("and now is ", @test = val, "\n")
end
```

`attr_reader(name, ...)`

Defines the reader method for the specified attribute(s).

`attr_writer(name, ...)`

Defines the writer method for the specified attribute(s).

`attr_accessor(name, ...)`

Defines both reader and writer methods for the specified attribute(s).

`ancestors`

Returns the list of the modules include in the receiver.

`class_eval(`*`src`*`)`
`class_eval{...}`

The aliass to the module_eval.

`constants`

Returns an array that holds names of the constants defined in the receiver.

`const_get(`*`name`*`)`

Returns the value of the specified constant. When specified constant is not defined, the `NameError` exception be raised.

`const_set(`*`name`*`, `*`value`*`)`

Defines the specified constant in the module. If the specified constant is already defined, the `NameError` exception is raised.

`extend_object(`*`object`*`)`

Append features (mothods and constants) to the specified object. `Object#extend` is defined using this method, so that redefining this method overrides extention behavior for the module.

`include(`*`module...`*`)`

Includes the modules specified to add methods and constants to the receiver module or class. `include` is for the Mix-in, which is disciplined multiple inheritance.

`included_modules`

Returns the list of the modules include in the receiver.

`instance_methods`

Returns the names of the public methods defined in the receiver.

`method_added(`*`id`*`)`

Will be called when a method defined for the receiver.

`method_defined?(`*`id`*`)`

Returns true, if the instance of the Module has the method specified by the *id*.

`module_eval(`*`expr`*`)`
`module_eval{...}`

Evaluates the *expr* string in the module's context. If block supplied for the method, the block is evaluated under the module's context.

In *module_eval()*'s context:

- *self*
- instance variables
- constants
- method definitions

are treated as if it appears in the module's definition body. But local variables are shared with *eval()*'s outer scope.

`module_function(`*name*`...)`

Makes the methods specified by *name*s into `module function's. the module functions are the method which is also the singleton method of a module (or a class). For example, methods defined in the Math module are the module functions.

`name()`

Returns the module's name.

`private(`*name*`...)`

If no argument is supplied, changes the default visibility in the class/method definition as private.

Examples:

```
module Foo
  def foo1() 1 end         # the default is the public
  private                  # the default changed to private
  def foo2() 2 end         # foo2 is the private method
end

foo = Foo.new
foo.foo1
       => 1
foo.foo2
       error--> private method `foo2' called #<Foo:0x4011ad8c>(Foo)
```

With the arguments, it makes the specified methods to be private.

`private_instance_methods`

Returns the names of the private methods defined in the receiver.

`protected(`*name*`...)`

If no argument is supplied, changes the default visibility in the class/method definition as public, where `protected' means the method can only be called from the method defined in the same class or its subclass. The check will be done in run-time, not compile-time.

With the arguments, it makes the specified methods to be protected.

`public [`*name*`...]`

If no argument is supplied, changes the default visibility in the class/method definition as public.

With the arguments, it makes the specified methods to be private.

Exapmles:

```
def foo() 1 end
foo
```

```
        => 1
self.foo # the toplevel default is private
        error--> private method `foo' called for "main"(Object)

def bar() 2 end
public :bar       # visibility changed (all access allowed)
bar
        => 2
self.bar
        => 2
```

`private_class_method(`*name*`, ...)`
`public_class_method(`*name*`, ...)`

Changes visibility of the class methods (class's singleton methods).

`remove_const(`*name*`)`

Removes definition of the named constant. Raises `NameError`, if specified constant is not defined in the module. Some constants defined in the `Object` class, like predefined classes are not allowed to be removed, so removing these constants raises `NameError` also.

`remove_method(`*name*`)`

Removes the named method from the module. Raises `NameError`, if specified method is not defined in the module.

`undef_method(`*name*`)`

Cancels the method definition specified by *name*, which is the string or ID (Integer).

Class Methods:

`nesting`

Returns the nesting of the class/module definitions at the calling point.

`new`

Creates an anonymous module.

Class

The class of the classes. To tell the truth, each class have the unnamed class (which is the *meta-class* of the class), the class `Class` is the class of these *meta-classes*. It is complicated, but it is not important for using Ruby.

SuperClass:

`Module`

Methods:

`ancestors`

Returns the list of the superclass and inclded modules with precedence.

`method_defined?(`*`id`*`)`

Returns true, if the instance of the Class has the method specified by the *id*.

`inherited(`*`subclass`*`)`

Will be called when a subclass of the receiver created. The argument is the newly created subclass.

`new(...)`

Creates an instance of the class. This arguments to this method will pass to the `initialize`.

`name()`

Returns the class name.

`superclass`

Returns the superclass of the class.

Class Methods:

`new([`*`superclass`*`])`

Creates an anonymous class, which superclass is specified by *superclass*. The default value for the *superclass* is the class `Object`.

Numeric

`Numeric` is the abstract class for the numbers.

SuperClass:

`Object`

Included Modules:

`Comparable`

Methods:

`+ self`

Returns `self`.

`- self`

Returns negation of `self`.

`self <=> other`

Returns -1, 0, or 1 depending on whether the left argument is numerically less than, equal to, or greater than the right argument.

`abs`

Returns absolute value of `self`.

`chr`

Returns the string that contains the character represented by that `Integer(self)` in the ASCII character set. For example, `65.chr` is `"A"`.

`coerce(`*`number`*`)`

Converts `self` and the *number* into mutually calculatable numbers.

`divmod(`*`other`*`)`

Returns a pair of the quotient and remainder by *other*.

`integer?`

Returns true if the receiver is integer.

`nonzero?`

Returns `false` on zero, the `self` otherwise.

`to_f`

Converts `self` into a floating-point number.

`to_i`

Returns an integer converted from `self`.

`zero?`

Returns `true` on zero.

Integer

The abstract class for integers, which has two subclasses `Fixnum` and `Bignum`. In Ruby, both kinds of integers are mixable, and converted automatically according to the value. Integers can be treated as infinite bit strings for bit operations.

SuperClass:

`Numeric`

Included Modules:

`Precision`

Class Methods:

`induced_from(`*num*`)`

Converts *num* into `Integer`.

Methods:

`self[`*nth*`]`

Returns 1 if *nth* bit of the integer set, otherwise 0.

`chr`

Returns the string contains a character represented by that number in the character set. For example, `65.chr` returns the string `"A"`.

The number must be within 0 to 255.

`downto(`*min*`) {...}`

Iterates from `self` to *min*, decrementing by 1.

`integer?`

Returns true.

`size`

Returns the approximate size of the integer in byte.

`step(`*max, step*`) {...}`

Iterates from `self` to *max*, adding *step* each time.

`succ`

Returns the ``next'' value of the integer.

`times {...}`

Iterates `self` times. `Self` may be rounded into integer.

`to_i`

Returns the receiver, since Integers do not need any conversion.

`upto(`*max*`) {...}`

Iterates from `self` to *max*, incrementing by 1.

Bignum

The class for long integers, limited only by memory size. Bignums converted into fixnums automatically if the value fits in, or vise versa. For the purpose of shift and mask operations, a binary representation is assumed, and negative numbers are represented in a 2's complement, which gives the illusion of an infinite string of sign bits extending to the left. Mixed calculation of Floats and Bignums, sometimes makes an error.

SuperClass

Integer

Methods:

`self + `*other*
`self - `*other*
`self * `*other*
`self / `*other*
`self % `*other*
`self ** `*other*

Arithmetic operators. Addition, subtraction, multiplication, division, remainder, power, respectively.

`~ self`
`self | `*other*
`self & `*other*
`self ^ `*other*

Bit manipulations. Bit reverse, bit or, bit and, bit xor, respectively.

`self << `*bits*
`self >> `*bits*

Shift operators, shift `bits`-bit to left and right, respectively.

`divmod(`*other*`)`

Returns an array contains the division and the modulo.

Fixnum

The class for the fixed range integers fit in the machine's pointer, which is 31 bits length on most machines. Fixnums are immediate values. If the result of the operations on fixnums exceed the range, it will automatically expanded to bignums.

SuperClass:

`Integer`

Methods:

`self + `*other*
`self - `*other*
`self * `*other*
`self / `*other*
`self % `*other*
`self ** `*other*

Arithmetic operations.

`~ self`
`self | `*other*
`self & `*other*
`self ^ `*other*
`self << `*bits*
`self >> `*bits*

Bit operations

`id2name`

Returns a corresponding strings to the number. If there is no corresponding symbol to the integer, it returns `nil`.

`remainder(other)`

Returns remainder of the numbers. In case either number is negative, the evaluated value differs from that of the operator `%`.

```
 13 %  4  =>  1
 13 % -4  => -3
-13 %  4  =>  3
-13 % -4  => -1
```

```
  13.remainder(4)    =>  1
  13.remainder(-4)   =>  1
(-13).remainder(4)   => -1
(-13).remainder(-4)  => -1
```

Float

The class of floating-point numbers

SuperClass:

Numeric

Included Modules:

Precision

Class Methods:

`induced_from(`*num*`)`

Converts *num* into `Float`.

Methods:

`self + `*other*
`self - `*other*
`self * `*other*
`self / `*other*
`self % `*other*
`self ** `*other*

Arithmetic operations.

`self == `*other*
`self < `*other*
`self <= `*other*
`self > `*other*
`self >= `*other*

Floating-point comparison.

`ceil`

Returns the smallest integer value not less than *self*.

`floor`

Returns the largest integer value not greater than *self*.

`round`

Rounds `self` to closest integer.

`to_f`

Returns the receiver, since Floats do not need any conversion.

Proc

The `Proc` is the procedure object, which is the encapsulated block with context, such as local variables and stack frames. The `Proc` object acts like an unnamed function except it does not have its own local variable scope. (Although, local variables which appear first in the block can have distinct value for each `Proc`.) The non local jump such as `return`, `break`, `next`, `redo`, `retry` raise the exceptions, once returned from `Proc` creation methods.

SuperClass:

`Object`

Class Methods:

`new`

Wraps the block with context and creates a new procedure object.

Methods:

`self[`*arg...*`]`
`call(`*arg...*`)`

Executes the procedure wrapped in the object. Arguments to the `call` are assigned to the block parameter.

Range

The `Range` class describes the interval. The range object is created by the range operator (`..` or `...`). The range object created by `..` operator include end. The range object created by `...` operator does not include end.

For example:

```
for i in 1..5
  ...
end
```

iterates over 1 to 5, including 5. On the other hand, `1...5` iterates over 1 through 4.

The both operands for the range operator must be comparable with `<=>` opeator. In addition they must have `succ` method for `each` operation.

Note That an interval contains both ends.

SuperClass:

`Object`

Included Modules:

`Enumerable`

Class Methods:

`new(`*`first`*`,`*`last`*`[, exclude_end])`

Creates the range object from *first* to *last*. Optional third argument specifies whether the range include *last* or not. If the third argument is ommitted, the range will include the *last*

Methods:

`self === `*`other`*

Returns true if `other` is within the range. This operator is mostly used by the `case` statement.

`each {|`*`item`*`| ...}`

Iterates over each item within the range.

`exclude_end?`

Returns true if the range will exclude the end.

`first`
`begin`

Returns the start point of the range.

`last`
`end`

Returns the end point of the range.

`length`
`size`

Returns length of the range (last - first + 1).

Regexp

The `Regexp` is the class for regular expression matching. Regular expression literals are enclosed by slashes like:

```
/^this is regexp/
```

In other way, you can create the regular expression object using:

```
Regexp.new(string)
```

SuperClass:

`Object`

Class Methods:

`compile(`*string*`[, `*casefold*`])`
`new(`*string*`[, `*options*`])`

Compiles the *string* into a regular expression object. If second argument given, and its value is true, then the created regexp object becomes case-insensitive. the value of the second argument is Fixnum, it must be bitwise-or of `Regexp::IGNORECASE` and `Regexp::EXTENDED`.

`quote(`*string*`)`

Insert escape characters before regular expression special characters in the `string`. Returns the newly created strings.

Methods:

`self =~ `*string*
`self === `*string*

Returns an index of the matching place in the string, if matched. Returns `nil` if not matched. The index of the first character is 0.

`~ self`

Matches with the value of the default variable (`$_`). Behaves like:

```
self =~ $_
```

`casefold?`

Returns true if the Regexp is compiled as case-insensitive.

`source`

Returns the orginal string form of the regular expression.

String

The `String` class represents the arbitrary length of byte string.

Some methods for the `String` class which have the name ends with `!', modify the contents of the strings. The methods without `!' make copy of the string and modify them. So, they are more slower but safer. For example:

```
f = "string"
print f, f.sub("str", "ski"), f
       => string, skiing, string
print f, f.sub!("str", "ski"), f
       => skiing, skiing, skiing
```

SuperClass:

`Object`

Included Modules:

`Comparable`
`Enumerable`

Class Methods:

`new(`*`string`*`)`

Returns a newly created string object which has same contents to the *string*.

Methods:

`self + `*`other`*

Returns newly created concatenated string.

`self * `*`times`*

Reputation of the string. For example, `"x" * 4` returns `"xxxx"`.

`self % `*`args`*

Formats from the string. Works just like: `sprintf(self, *args)`.

```
self == other
self > other
self >= other
self < other
self <= other
```

String comparison. If the value of the variable `$=` is not false, comparison done by case-insensitive.

```
self === other
```

Same as operator `==`. Used for `case` comparison.

```
self << other
```

Appends *other*'s value to the string contents.

```
self =~ other
```

String match. Returns an index of the match if any, or `nil`. If the argument *other* is a string, it will be compiled into a regular expression.

```
~ self
```

Same as `$_ =~ self`.

```
self[nth]
```

Retrieves the *nth* character from a string.

```
self[start..last]
```

Returns a substring from *start* to *last*, including both ends.

```
self[start, length]
```

Returns a substring of *length* characters from *start*.

```
self[nth] = val
```

Changes the *nth* character of the string into *val*. If the strings is freezed, exception will occur.

```
self[start..last] = val
```

Replace the substring from *start* to *last* with *val*.

```
self[start, len] = val
```

Replace the substring *length* characters from *start* with *val*.

```
self <=> other
```

Returns -1, 0, or 1 depending on whether the left argument is less than, equal to, or greater than the right argument in the dictionary order.

`<<(`*other*`)`
`concat(`*other*`)`

Appends the contents of the *other*.

`capitalize`
`capitalize!`

Changes the first character in the string to uppercase character, if it is an alphabet.

`capitalize!` returns nil, if it does not modify the receiver.

`chop`
`chop!`

Chops off the last character of the string (2 characters if the last characters are "\r\n"). `chop!` modifies the receiver.

`chop!` returns nil, if it does not modify the receiver.

`chomp([`*rs*`])`
`chomp!([`*rs*`])`

Chops off the line ending of the string, which is specified by the *rs*. The default value of the `rs` is the value of the variable `$/`.

`chomp!` modifies the receiver, and returns nil, if it does not modify the receiver.

`clone`
`dup`

Returns a newly created string object which has the save value to the string. `clone` returns the complete copy of the original string including freeze status and instance variables. On the other hand, `dup` copies the string contents only.

`crypt(`*salt*`)`

Returns an encoded string using `crypt(3)`. *salt* is the arbitrary string longer than 2 bytes.

`delete(`*str*`)`
`delete!(`*str*`)`

Deletes every characters included in `str` from the string. Deleting characters in *str* is in the form of `tr(1)`. For example, `` `a-c' `` is the range from `` `a' `` to `` `c' ``, `` `^' `` at the beginning means complement of the character set.

`delete!` returns nil, if it does not modify the receiver.

`downcase`
`downcase!`

Replaces all uppercase characters to lowercase characters. Little bit faster than `tr("A-Z", "a-z")`.

`downcase!` returns nil, if it does not modify the receiver.

`dump`

Replaces all non-printable characters into backslash notations. The condition `str == eval (str.dump)` is guaranteed.

`each_byte {|`*byte*`|...}`

Iterates over each byte of the string.

`each([`*rs*`]) {|`*line*`|...}`
`each_line([`*rs*`]) {|`*line*`|...}`

Iterates over each line in the string. The value of the *rs*is the line separator, which default is the value of the variable `$/`.

`empty?`

Returns true, if the string is empty (i.e. 0 length string).

`freeze`

Prohibits modification of the string. Modification to the freezed string raises an exception.

`frozen?`

Returns true if the string is frozen.

`gsub(`*pattern*`, `*replace*`)`
`gsub!(`*pattern*`, `*replace*`)`

Replaces all matching substrings with the *pattern* to the *replace*. In *replace*, `&' and `\0' replaced to the matching substring, `\digit' replaced to the contents of the corresponding parenthesis. `\``, `\'`, and `\+` replaced to pre-match, post-match substring, and contents of the last parenthesis respectively.

Notice: `$<digits>` are not available for the *replace* string, since it is evaluated **before**, match happens.

The method `gsub!` modifies the original string. If no match found `gsub!` returns `nil`. On the other hand, `gsub` modifies the copy of the string. If no match found, `gsub` returns the unmodified original string.

`gsub(`*pattern*`) {...}`
`gsub!(`*pattern*`) {...}`

If `gsub` and `gsub!` are called with the block, replace all matching substring to the value of the block. The matching substring will be given to the block.

`hex`

Interprets the string as a hexadecimal string and returns the corresponding integer value.

`index(`*substr*`[, `*pos*`])`

Returns the index of the *substr* in the string, or `nil` if not found. Optional second argument *pos* is given, start lookup from there.

`intern`

Returns unique integer corresponding the string. The string must not contain the null character ('\0').

`length`
`size`

Returns the length of the string in bytes.

`ljust(`*`width`*`)`
`rjust(`*`width`*`)`
`center(`*`width`*`)`

Returns left filled, right filled, centered string, respectively. If the string is longer than *width*, returns the string itself without modification.

`oct`

Interprets the string as a octal string and returns the corresponding integer value. Returns 0 if the string is not in octal. The octal string is the pattern of `/^[0-7]+/`.

`reverse`
`reverse!`

Returns the reversed string of the original string.

`replace(`*`other`*`)`

Copis the content of *other* into the string.

`rindex(`*`substr`*`[, `*`pos`*`])`

Returns the index of the last occurrence of the *substr*, or `nil` if not found. If optional second argument *pos* given, `rindex` returns the last occurrence at or before that position.

`scan(`*`pattern`*`)`
`scan(`*`pattern`*`) {...}`

Returns the array of arrays which contain subpatterns corresponding parentheses in the *pattern*. Match will be done repeatedly to the end of the string. When called with the block, subpatterns are passed to the block as parameters.

`split([`*`sep`*`[, `*`limit`*`]])`

Return an array containing the fields of the string, using the string or regexp *sep* as a separator. If *sep* is omitted, the value of the variable `$;` will be used as default. If the value of `$;` is `nil`, splits the string on whitespace (after skipping any leading whitespace). Anything matching *sep* is taken to be a delimiter separating the fields. (Note that the delimiter may be longer than one character.)

If *limit* is unspecified, trailing null fields are stripped (which potential users of pop() would do well to remember). If *limit* is specified and is not negative, splits into no more than that many fields (though it may split into fewer). If *limit* is negative, it is treated as if an arbitrarily large *limit* had been specified.

A pattern matching the null string (not to be confused with a null pattern `//`, which is just one member of the set of patterns matching a null string) will split the string into separate characters at each point it matches that way. For example:

```
print 'hi there'.split(/ */).join(':');
```

produces the output 'h:i:t:h:e:r:e'.

```
squeeze([str])
squeeze!([str])
```

Squeezes sequences of the same characters which is included in the *str*.

`squeezes!` returns nil, if it does not modify the receiver.

```
strip
strip!
```

Removes leading and trailing whitespace from the string.

`strip!` returns nil, if it does not modify the receiver.

```
sub(pattern, replace)
sub!(pattern, replace)
```

Replaces the first matching substrings with the *pattern* to the *replace*. In *replace*, `&' and `\0' replaced to the matching substring, `\digit' replaced to the contents of the corresponding parenthesis.

Notice: `$<digits>` are not available for the *replace* string, since it is evaluated **before**, match happens.

The method `sub!` modifies the original string. If no match found, `sub!` returns `nil`. On the other hand, `sub` modifies the copy of the string. If no match found, `sub` returns the unmodified original string.

```
sub(pattern) {...}
sub!(pattern) {...}
```

If `gsub` and `gsub!` are called with the block, replace the first matching substring to the value of the block. You can use `$<digits>` in the iterator block, for the block is evaluated **after** the match. The matching substring will be given to the block.

```
succ
succ!
```

Returns the **succeeding** string from `self`, which is like:

```
"aa".succ =>      "ab"
"99".succ =>      "100"
"a9".succ =>      "b0"
"Az".succ =>      "Ba"
"zz".succ =>      "aaa"
```

```
sum([bits])
```

Calculates *bits*-bit checksum of the string. Default is 16-bit checksum. For example, the following computes the same number as the System V sum program:

```
sum = 0
while gets
  sum += $_.sum
end
sum %= 65536
```

```
swapcase
swapcase!
```

Replaces all lowercase characters to uppercase characters, and all uppercase characters to lowercase characters.

`swapcase!` returns nil, if it does not modify the receiver.

`to_f`

Converts the string into `Float`.

`to_i`

Interprets the string as a decimal string and returns the corresponding integer value.

`tr(`*`search, replace`*`)`
`tr!(`*`search, replace`*`)`

Translates all occurrences of the characters found in *search*, with the corresponding character in *replace*. The first character in the *search* is `^', characters not in the *search* are translated.

`tr!` returns nil, if it does not modify the receiver.

`tr_s(`*`search, replace`*`)`
`tr_s!(`*`search, replace`*`)`

Translates all occurrences of the characters found in *search*, with the corresponding character in *replace*, then squeezes sequences of the same characters within the replaced characters. The first character in the *search* is `^', characters not in the *search* are translated.

`tr_s!` returns nil, if it does not modify the receiver.

`unpack(`*`template`*`)`

Unpacks packed string data (probably made by `Array#pack`), and expanded array value. The *template* has the same format as in the `Array#pack`, as follows this:

`a`	ASCII string(null padded)
`A`	ASCII string(space padded)
`b`	bit string(ascending bit order)
`B`	bit string(descending bit order)
`h`	hex string(low nibble first)
`H`	hex string(high nibble first)
`c`	char
`C`	unsigned char
`s`	short
`S`	unsigned short
`i`	int
`I`	unsigned int
`l`	long
`L`	unsigned long
`m`	string encoded in base64
`n`	short in "network" byte-order
`N`	long in "network" byte-order
`v`	short in "VAX" (little-endian) byte-order
`V`	long in "VAX" (little-endian) byte-order
`f`	single-precision float in the native format
`d`	A double-precision float in the native format
`p`	A pointer to a null-terminated string.
`P`	A pointer to a structure (fixed-length string).
`u`	uuencoded string

`x`	skip a byte
`X`	back up a byte
`@`	moves to absolute position

`upcase`
`upcase!`

Replaces all lowercase characters to downcase characters. Little bit faster than `tr("a-z", "A-Z")`.

`upcase!` returns nil, if it does not modify the receiver.

`upto(`*`max`*`) {...}`

Iterates from `self` to *max*, giving the **next** string each time. (See `succ`) This method used internally in `Range#each`, that makes

```
for i in "a" .. "ba"
  print i, "\n"
end
```

print `a, b, c,...z,aa,...az, ba'.

Struct

The structure class. `Struct.new` creates the new subclass of the `Struct` class and returns it. Each subclass has access methods for the structure attributes.

SuperClass:

`Object`

Included Modules:

`Enumerable`

Class Methods:

`new(`*`name, member...`*`)`

Creates the new subclass of the `Struct` class, which is named *name* and returns it. Each subclass has access methods for the structure attributes. For example:

```
dog = Struct.new("Dog", :name, :age)
fred = dog.new("fred", 5)
fred.age=6
printf "name:%s age:%d", fred.name, fred.age
```

prints `"name:fred age:6"`.

The structure *name* should start with captal letter, since it is class constant name under the `Struct` class.

Class Methods for each structure:

`new(`*`value...`*`)`
`[`*`value...`*`]`

Creates a structure. The arguments are initial value of the structure. The number of arguments must be same to the number of attributes of the structure.

`members`

Returns an array of the struct member names.

Methods:

`self[`*`nth`*`]`

Returns the value of the *nth* attribute. If *nth* is string, returns the value of the named attribute.

`each`

Iterates over each value of the struct member.

`members`

Returns an array of the struct member names.

`values`
`to_a`

Returns the attributes of the structure as an array. For example, the following code prints your passwd entry.

```
print Etc.getpwuid.values.join(":"), "\n"
```

Time

The `Time` class represents the time value and its operations. `Time.now` returns the current time object. The timestamps of the files returned by `File#stat` are instances of this class.

SuperClass:

`Object`

Included Modules:

`Comparable`

Class Methods:

`now`
`now`

Returns the current time object.

`at(`*`time`*`)`

Creates the time object at *time*. The argument *time* must be either the instance of the `Time` class or the number, which is considered as seconds since `00:00:00 GMT, January 1, 1970`.

`gm(`*`year, month, day, hour, min, sec`*`)`

Returns the `Time` object specified by the arguments in GMT. The arguments after the second one can be omitted. The default value for the omitted argument is the lowest value for that argument.

If the `month` argument is a fixnum, it must be from 1(January) to 12(December). If it is string, it must be the English month name or the number starting from 1(January) to 12(December).

If the number of arguments are equals to that of elements of Time#to_a, `Time.gm` understands them properly, even if they are in the little endian (seconds comes first) format.

`local(`*`year, month, day, hour, min, sec`*`)`
`mktime(`*`year, month, day, hour, min, sec`*`)`

Returns the `Time` object specified by the arguments in localtime. The arguments after the second one can be omitted. The default value for the omitted argument is the lowest value for that argument.

`times`

Returns the user and system CPU times structure, which attributes are:

```
utime             # user time
stime             # system time
cutime            # user time of children
cstime            # system time of children
```

Times are floating-point numbers and are measured in seconds. (See `times(3)`)

Methods:

`self + `*`other`*

Returns the time object which is later than `self` by *other*.

`self - `*`other`*

Returns the diff of the time in float, if the *other* if the time object. If the *other* is the numeric value, it returns the time object earlier than `self` than *other*.

`self <=> `*`other`*

Comparison of the times. *other* must be a time object or an integer or a floating-point number. Numbers are considered as seconds since `00:00:00 GMT, January 1, 1970`.

`asctime`
`ctime`
`to_s`

Converts the time into the string form like `ctime(3)`.

`gmtime`

Sets timezone of the time object to GMT. The time object operates time in GMT thereafter. To print time in GMT:

```
print Time.now.gmtime, "\n"
```

`gmtime` returns `self`.

`localtime`

Sets timezone of the time object to localtime, which is default. Returns time object itself.

`to_i`
`tv_sec`

Returns the time since the epoch, `00:00:00 GMT, January 1, 1970`, measured in seconds.

`sec`
`min`
`hour`
`mday`
`day`
`mon`
`month`
`year`
`wday`
`yday`
`zone`
`isdst`

Returns the internal values of the time object. All methods return an integer value, except `zone`, which returns the timezone name string. (c.f. `localtime(3)`)

Notice: Unlike tm struct, month returns 1 for January, year returns 1998 for year 1998, and yday start with 1 (not 0).

`strftime(`*`format`*`)`

Returns formatted string from the time object. Format specifiers are as follows:

`%A`	full weekday name(Sunday, Monday...)
`%a`	abbreviated weekday name(Sun, Mon...)
`%B`	full month name(January, February...)
`%b`	abbreviated month name(Jan, Feb...)
`%c`	date and time representation
`%d`	day of the month in decimal(01-31)
`%H`	hour using a 24-hour clock(00-23)
`%I`	hour using a 12-hour clock(01-12)

%j	day of the year(001-366)
%M	minutes(00-59)
%m	month in decimal(01-12)
%p	Either AM or PM
%S	second in decimal(00-61)
%U	week number, first Sunday as the first day of the first week(00-53)
%W	week number, first Monday as the first day of the first week(00-53)
%w	day of the week in number. Sunday being 0(0-6)
%X	time representation without date
%x	date representation without time
%Y	year
%y	year without century(00-99)
%Z	timezone
%%	%itself

`to_a`

Converts tm struct into an array, whose elements are in order of:

- sec
- min
- hour
- mday
- mon
- year
- wday
- yday
- isdst
- zone

`to_f`

Converts the time (in seconds from the epoch) into a floating-point number.

`usec`
`tv_usec`

Returns micro second part of the time object.

NilClass

The class of `nil`. The `nil` is the sole instance of this class. The `nil` is false, along with the `false` object. Any other objects are true.

Notice: This class is for internal use only. The constant `Nil` is not defined, and can not be accessed from Ruby scripts.

SuperClass:

`Object`

Comparable

`Comparable` is the Mix-in for comparable classes. The including class must provide the basic comparison operator `<=>'. The other comparing operators are defined in this module using operator `<=>'.

Methods:

`self == other`

Returns true if `self` and the *other* have same values.

`self > other`

Returns true if `self` is greater than the *other*.

`self >= other`

Returns true if `self` is greater than or equals to the *other*.

`self < other`

Returns true if `self` is less than the *other*.

`self <= other`

Returns true if `self` is less than or equals to the *other*.

`between?(min, max)`

Returns true if *self* is in between *min* and *max*.

Enumerable

Enumerable is the Mix-in module for the enumeration. The including class must provide the method `each`. All methods provided by `Enumerable` are defined using `each`.

Methods:

`collect {|item|...}`

Returns an array of the result of the block evaluation over each item.

`each_with_index {|item,index|...}`

Iterates over each element with its index.

```
find {|item|...}
detect {|item|...}
```

Returns the first item which satisfies the block condition (i.e. the block's return value is true).

```
find_all {|item|...}
select {|item|...}
```

Returns an array of all items which satisfy the block condition.

```
grep(pattern)
grep(pattern) {|item|...}
```

Returns an array of all items which satisfy `*pattern*=== item'. If is called with the block, `grep` evaluates the block over every item matched.

```
member?(val)
include?(val)
```

Returns true if there is an item which equals to *val*. Comparison is done by the operator `==='.

```
index(val)
```

Returns the index of the item which equals to *val* using operator `==='. The index of the first item is 0. Returns `nil` if there is no matching item. It is meaningless for non-ordered enumerables.

```
length
size
```

Returns the number of items.

```
min
```

Returns the smallest item assuming all items are `Comparable`.

```
min{|a, b|...}
```

Returns the smallest item using the evaluated value of the block.

```
max
```

Returns the greatest item assuming all items are `Comparable`.

```
max{|a, b|...}
```

Returns the greatest item using the evaluated value of the block.

```
reject {|item|...}
```

Returns an array of all items which does not satisfy the block condition.

```
sort
sort {|a, b|...}
```

Returns the sorted array of the items. If the block is given, it must compare two items just like <=>.

`to_a`
`entries`

Converts an Enumerable to an array.

`---`

Errno

The module for the system call exceptions.

Errno::E*

The exception classes corresponding each `errno`. For the detail, see `man errno`.

SuperClass:

`SystemCallError`

Constants:

`E*::Errno`

The `errno` value for the exception.

FileTest

`FileTest` is the collection of the file test functions. It can be used for inclusion.

Module Functions:

`blockdev?(`*`file`*`)`

Returns true if the file specified by the *file* is the block special file.

`chardev?(`*`file`*`)`

Returns true if the file specified by the *file* is the character special file.

`executable?(`*`filename`*`)`

Returns true if the file specified by the *filename* is executable by the effective user/group id.

`executable_real?(`*`filename`*`)`

Returns true if the file specified by the *filename* is executable by the real user/group id.

`exist?(`*`file`*`)`

Returns true if the file specified by the *file* exists.

`grpowned?(`*`file`*`)`

Returns true if the file specified by the *file* has gid of effective group.

`directory?(`*`file`*`)`

Returns true if the file specified by the *file* is a directory.

`file?(`*`file`*`)`

Returns true if the file specified by the *file* is a regular file.

`pipe?(`*`file`*`)`

Returns true if the file specified by the *file* is a named pipe (FIFO).

`socket?(`*`file`*`)`

Returns true if the file specified by the *file* is a socket.

`owned?(`*`file`*`)`

Returns true if the file specified by the *file* is owned by you.

`readable?(`*`filename`*`)`

Returns true if the file specified by the *filename* is readable by you.

`readable_real?(`*`filename`*`)`

Returns true if the file specified by the *filename* is readable by your real uid/gid.

`setuid?(`*`file`*`)`

Returns true if the setuid bit of the file specified by the *file* is set.

`setgid?(`*`file`*`)`

Returns true if the setgid bit of the file specified by the *file* is set.

`size?(`*`file`*`)`

Returns the size of the file specified by the *file*. Returns `nil` if it does not exist or is empty.

`sticky?(`*`file`*`)`

Returns true if the sticky bit of the file specified by the *file* is set.

`symlink?(`*`file`*`)`

Returns true if the file specified by the *file* is a symbolic link.

`writable?(`*`filename`*`)`

Returns true if the file specified by the *filename* is writable by you.

`writable_real?(`*`filename`*`)`

Returns true if the file specified by the *filename* is writable by your real uid/gid.

`zero?(`*`file`*`)`

Returns true if the file specified by the *file* exists, and the size of the file is 0.

GC

GC is the module to control garbage collection feature of the Ruby interpreter.

Methods:

`garbage_collect`

Starts garbage collecting. Same as

```
GC.start
```

Module Methods:

`disable`

Disables garbage collection.

`enable`

Enables garbage collection.

`start`

Start garbage collection.

Kernel

The module included by the `Object` class, which defines the methods available for all classed. The methods described in the built-in functions section are defined in this module.

Marshal

The module for reading and writing Ruby object in a binary format. This format is machine independent. It can dump most of Ruby objects, there are several exceptional objects like `IOs`. Dumping such objects raises TypeError exception.

module functions:

`dump(`*obj*`[,`*port*`][,`*limit*`])`

Dumps object recursively. Dumping some kind of instances of the classes like Class, Module, IO, Data, etc. will raises TypeError exception.

*port*must be the instance of the `IO` or its subclass. If *port* is ommited `dump` returns the dumped image string.

If the dumping object defines `` `_dump' `` method, *dump* calls that method to dump with the limit argument. It must return the dumped image string. If `` `_dump' `` method is provided, the class must provide the reverse method named `` `_load' `` as its singleton method, which is called with dumped image string, and should return the retsored object.

The *limit* argument specified the level of the recursive traverse. the default limit value is 100. In case negative limit value is specified, no limit will checked for dumping.

`load(`*port*`[, `*proc*`])`

Creates new object from *port*, which is either IO or String. If optional second argument *proc* is specified, the proc will called with every created object as an argument. It can be used, for example, schema evolution.

Math

The modules for the floating point arithmetic. The *Math* module has methods and modules methods of the

same definitions in same names.

Example:

```
pi = Math.atan2(1, 1)*4;
include Math
pi2 = atan2(1, 1)*4
```

Module Functions:

`atan2(`*y*`, `*x*`)`

Returns the arc tangent of the two variable *x* and *y* in radian, which is between -PI and PI.

`cos(`*x*`)`
`sin(`*x*`)`
`tan(`*x*`)`

Returns the cosine, sine, tangent, respectively, of x, where x is given in radians, which is between -1 and 1.

`exp(`*x*`)`

Returns the value of *e* raised to the power of *x*.

`frexp(`*x*`)`

Returns the nomalized fraction and an the exponent.

`ldexp(`*x*`,`*exp*`)`

Returns the result of multiplying the floating-point number x by 2 raised to the power of *exp*.

`log(`*x*`)`

Returns the natural logarithm of *x*.

`log10(`*x*`)`

Returns the base-10 logarithm of *x*.

`sqrt(`*x*`)`

Returns non-negative square root of *x*. It raises an exception, if *x* is negative.

Constants:

`PI`

The pi.

`E`

The e.

ObjectSpace

The module to iterate over living objects.

Module Functions:

`add_finalizer(`*`proc`*`)`

Sets the *proc* as the finalizer. When the object that specified by `call_finalizer` are going to be recycled, the finalizers are called with their ID's (c.f Object#id).

`call_finalizer(`*`obj`*`)`

Set the finalizer flag for the object. The finalizers are called when objects that the flag set are going to recycle.

`finalizers`

Returns the list of the finalizers.

`each_object([`*`class_or_module`*`])`

Iterates over all living objects which is instance of the *class_or_module*. Without an argument, it iterates over all existing objects.

`garbage_collect`

Starts collecting objects which is no longer accessed from anywhere.

`remove_finalizer(`*`proc`*`)`

Removes the *proc* from the finalizers.

---

prec

Precision

`Precision` is a `Mix-in` for concrete numerical classes with precision. Here, `precision' means the fineness of approximation of a real number, so, this module should not be included into non-subset of real, e.g. abstract numerical, complex, matrix.

Class Methods:

`induced_from(`*`number`*`)`

Creates an ojbect which is converted from *`number`*. Since it raise TypeError in default, redefine before uses. Note that a use of `prec` in a redefinition may causes an infinite loop.

Method:

`prec(`*`klass`*`)`

Converts `self` into a object of *klass*. In default, `prec` invokes *`klass`*`.induced_from(self)` and returns its value. So, if *`klass`*`.induced_from` doesn't correspond to the class of self, it is neccessary to redefine this `prec`.

`prec_i`

Returns an integer converted from `self`. It is equivalent to `prec(`*`Integer`*`)`.

`prec_f`

Returns a floating-point number converted from `self`. It is equivalent to `prec(`*`Float`*`)`.

Process

The module for operating (UNIX) processes. Like `Math` module, all methods defined in this module are module functions.

Notice the `Process` is not a class for process object, but the module for methods to operate processes.

Module Attributes:

`egid`

Returns the current effective group ID of the process.

`egid= `*`gid`*

Sets the current effective group ID of the process to *gid*.

`euid`

Returns the current effective user ID of the process.

`euid= `*`uid`*

Sets the current effective user ID of the process to *uid*.

`gid`

Returns the current real group ID of the process.

`gid= gid`

Sets the current real group ID of the process to *gid*.

`pid`

Returns the process ID of the process, which is same as the value of the variable `$$'.

`ppid`

Returns the process ID of the parent process. Under UN*X, the process ID will be 1 (the pid of `init`) after the parent process terminates.

`uid`

Returns the real user ID of the process.

`uid= uid`

Sets the real user ID of the process to *uid*.

Module Functions:

`getpgrp([pid])`

Returns the current process group for the *pid*, 0 for the current process. If PID is omitted, returns process group of current process.

`getpriority(which, who)`

Returns the current priority for a process, a process group, or a user. (See `getpriority(2)`.) The Process module defines the constants for *which*, *PRIO_PROCESS*, *PRIO_PGRP*, *PRIO_USER*.

`kill(signal, pid...)`

Sends *signal* to the specified *pid*. *signal* must be a string or a integer to specify the signal to send. If the signal is negative (or `-` before the signal name), it kills process groups instead of processes.

`setpgrp(pid, pgrp)`

Sets the current process group for the *PID*, 0 for the current process.

`setpriority(which, who, prio)`

Sets the current priority for a process, a process group, or a user. (See `setpriority(2)`.) The Process module defines the constants for *which*, *PRIO_PROCESS*, *PRIO_PGRP*, *PRIO_USER*.

`setsid()`

Creates a new session, and detaches tty. Returns the session ID of the new session.

`wait`

Waits for a child process to terminate and returns the pid of the deceased process, or raise

`Errno::ECHILD` if there are no child processes.

`waitpid(`*`pid, flags`*`)`

Waits for a particular child process specified by *pid* to terminate and returns the pid of the deceased process, or raise `Errno::ECHILD` if there is no such child process. Returns `nil` if the process did not terminated yet in the non-blocking mode. Non-blocking wait is only available on machines supporting either the `waitpid(2)` or `wait4(2)` system calls. However, waiting for a particular pid with FLAGS of `nil` or 0 is implemented everywhere.

Constants:

`PRIO_PROCESS`

Specifies process priority for `getpriority`/`setpriority`.

`PRIO_PGRP`

Specifies process group priority.

`PRIO_USER`

Specifies user priority.

`WNOHANG`

Means to return immediately if no child has exited.

`WUNTRACED`

Means to also return for children which are stopped, and whose status has not been reported.

Bundled Libraries

Sorry, the document for the bundle libraries are not yet available except for Thread.

- **dbm**
- **etc**
- **kconv**
- **getopts**
- **parsearg**

- **parsedate**
- **socket**
  - BasicSocket
    - IPSocket
      - TCPSocket
        - TCPServer
      - UDPSocket
    - UNIXSocket
      - UNIXServer
    - Socket
- **thread**
  - Thread
  - Mutex
  - Queue
- **tk**

  The documentation of the Ruby Tk interface is not yet available. The classes in the Tk interface are shown below.

  - TkObject
    - TkWindow
      - TkRoot
      - TkToplevel
      - TkFrame
      - TkLabel
        - TkButton
          - TkRadioButton
            - TkCheckButton
        - TkMessage
        - TkMenuButton
      - TkScale
      - TkScrollbar
      - TkTextWin
        - TkListbox
      - TkMenu
      - TkScale
      - TkScale
      - TkScale
    - TkcItem
      - TkcArc
      - TkcBitmap
      - TkcImage
      - TkLinec
      - TkcOval
    - TkImage
      - TkBitmapImage
      - TkPhotoImage

DBM

`DBM` is the class to provide access to the DBM files. It acts string to string dictionary, storing data in the

DBM file.

SuperClass:

Object

Included Modules:

Enumerable

Class Method:

open(*dbname*[, *mode*])

Opens the DBM database specified by *dbname*, with *mode*. The default value for the *mode* is 0666. If nil is specified for the *mode*, open returns nil, if the DBM file does not exist.

Method:

self[*key*]

Returns the value corresponding the *key*.

self[*key*] = *value*

Stores the *value* associating with *key*. If nil is given as the *value*, the association from the *key* will be removed.

clear

Makes the DBM file empty.

close

Closes the DBM file. Later operation raises an exception.

delete(*key*)

Removes the association from the *key*.

delete_if { |*key*, *value*| ... }

Deletes associations if the evaluation of the block returns true.

each {|*key*, *value*| ... }
each {|*key*, *value*| ... }

Iterates over associations.

each_key {|*key*| ... }

Iterates over keys.

each_value {|*value*| ... }

Iterates over values.

`empty?()`

Returns true if the database is empty.

`has_key?(`*key*`)`
`key?(`*key*`)`
`include?(`*key*`)`

Returns true if the association from the *key* exists.

`has_value?(`*value*`)`
`value?(`*value*`)`

Returns true if the association to the *value* exists.

`indexes(`*key_1*`,...)`
`indices(`*key_1*`,...)`

Returns an array contains items at each key specified by each argument.

`keys`

Returns the array of the keys in the DBM dictionary.

`length`
`size`

Returns the number of association in the DBM dictionary. Current implementation uses iteration over all associations to count, so that it may take time to get the size of the huge DBM dictionary.

`shift`

Removes and returns an association from the database.

`values`

Returns the array of the values in the DBM dictionary.

Kconv module

Kconv - Kanji code conversion utility module

SYNOPSIS

```
newstring = Kconv::kconv(string, Kconv::JIS, Kconv::AUTO);
newstring = Kconv::tojis(string);
newstring = Kconv::toeuc(string);
newstring = Kconv::tosjis(string);
guessed_code = Kconv::guess(string);
```

CONSTANTS

AUTO
: Auto detection(only for *input* mode)

JIS
: ISO-2022-JP

EUC
: EUC-Japan

SJIS
: Shifted JIS code (so called MS Kanji code)

BINARY
: Binary code (not in JIS/SJIS/EUC).

UNKNOWN
: Couldn't determin character code. Same as AUTO.

Socket::Constants

The modules that contains the constants (such as `AF_INET`) required for the socket operations.

BasicSocket

The abstract class for the sockets. The operations are defined in the subclasses. For instance `TCPSocket` for the Internet domain stream socket.

SuperClass:

`IO`

Methods:

`getsockname`

Returns the information of the socket in the sockaddr structure packed in the string. See `getsockname(2)` for detail.

`getsockopt(`*`level`*`, `*`optname`*`)`

Get options of the socket. See `getsockopt(2)`. Returns the option data packed in the string.

`getpeername`

Returns the information of the peer connected to the socket. Returns the sockaddr structure packed in the string. See `getpeername(2)` for detail.

`recv(`*`len`*`[, `*`flags`*`])`

Receives data from the socket and returns as an string. The *len* is the maximum length of the receiving data. See `recv(2)`. The default value for the flags is 0. The constants for *flags* are defined in `Socket` class (ex. `Socket::SO_LINGER`). It bypasses stdio, so mixing this with other kinds of reads/eof checks may cause confusion.

`send(`*`mesg`*`, `*`flags`*`[, `*`to`*`])`

Sends the *mesg* through the socket. See `send(2)` for detail. You have to specify the *to* argument for the unconnected socket. Reurns the length of data sent.

`setsockopt(`*`level`*`, `*`optname`*`, `*`optval`*`)`

Sets socket options. See `setsockopt(2)` for detail.

`shutdown([how])`

Causes the connection of the socket to be shut down. If *how* is 0, further receives will be rejected. If *how* is 1, further sends will be rejected. If *how* is 2, further sends and receives will be rejected. The default value for the *how* is 2. See `shutdown(2)`.

IPSocket

The class for the Internet domain socket.

SuperClass:

`BasicSocket`

Class Methods:

`getaddress(host)`

Returns the address of the host. The address is an octet decimal string.

Methods:

`addr`

Returns the array contains socket connection information. The first element is the string `"AF_INET"`. The second element is the port number. The third element is the host representing string. The fourth element is the IP address of the host in the octet decimal string.

`peeraddr`

Returns the array contains the peer's socket information. The elements of the array is same as the ones which `addr` returns.

TCPSocket

The class for the Internet domain stream socket. The socket programming will be much easier using this class. For example, the socket client that sends the user input will be:

```
require "socket"

port = if ARGV.size > 0 then ARGV.shift else 4444 end
print port, "\n"

s = TCPSocket.open("localhost", port)

while gets
  s.write($_)
  print(s.gets)
end
s.close
```

SuperClass:

`IPSocket`

Class Methods:

`open(`*`host`*`, `*`service`*`)`
`new(`*`host`*`, `*`service`*`)`

Creates and returns the socket connected to the specified *service* on the *host*. The *host* is the host name string. The *service* is the service name registered in the `/etc/services` (or in NIS), or the port number.

`gethostbyname(`*`host`*`)`

Returns the array conaining the host information from the host name or the IP address (32 bit integer or string such as `"127.0.0.1"`). The first element of the array is the hostname. The second element is the array of the host aliases (possibly empty). The third element is the address type. And sequence of the addresses follows. The addresses are octet decimal string (like `"127.0.0.1"`).

Methods:

`recvfrom(`*`len`*`[, `*`flags`*`])`

Receives data from the socket and returns the pair of data and the address of the sender. Refer IPSocket#addr for the address format. For arguments, see `recv`.

TCPServer

The server side of the Internet stream socket. This class makes building server much easier. For example, the echo server will be:

```
require "socket"

gs = TCPServer.open(0)
socks = [gs]
addr = gs.addr
addr.shift
printf("server is on %d\n", addr.join(":"))

while true
  nsock = select(socks)
  next if nsock == nil
  for s in nsock[0]
    if s == gs
      socks.push(s.accept)
      print(s, " is accepted\n")
    else
      if s.eof?
         print(s, " is gone\n")
         s.close
         socks.delete(s)
      else
         str = s.gets
         s.write(str)
      end
    end
  end
end
```

Even shorter using thread:

```
require "socket"

gs = TCPServer.open(0)
addr = gs.addr
addr.shift
printf("server is on %d\n", addr.join(":"))

while true
  ns = gs.accept
  print(ns, " is accepted\n")
  Thread.start do
    s =  ns                         # save to dynamic variable
    while s.gets
      s.write($_)
    end
    print(s, " is gone\n")
    s.close
  end
end
```

SuperClass:

TCPSocket

Class Methods:

`new([host, ]service)`
`open([host, ]service)`

Opens new server connection. The *service* is the service name registered in the `/etc/services` (or in NIS), or the port number. If *host* is given, only the connection from the specified host will be accepted. If *host* is not specified, connection from any host will be accepted.

Methods:

`accept`

Accepts the request for the connection, and returns the `TCPSocket` connected to the client.

UDPSocket

The UDP/IP datagram socket class.

SuperClass:

IPSocket

Class Methods:

`open()`
`new()`

Returns new UDP socket.

Methods:

`bind(`*`host, port`*`)`

Binds the socket to the *port* on the *host*.

`connect(`*`host, port`*`)`

Connects the socket to the *port* on the *host*.

`recvfrom(`*`len`*`[, `*`flags`*`])`

Receives data from the socket and returns the pair of data and the address of the sender. Refer IPSocket#addr for the address format. For arguments, see `recv`.

`send(`*`mesg, flags[, host, port`*`])`

Sends the *mesg* through the socket. See `send(2)` for detail. You have to specify the *host* and *port* arguments for the unconnected socket. Reurns the length of data sent.

UNIXSocket

The UNIX domain stream socket.

SuperClass:

`BasicSocket`

Class Methods:

`open(`*`path`*`)`
`new(`*`path`*`)`

The socket associated to the *path*.

Methods:

`addr`

Returns the array contains socket connection information. The first element is the string `"AF_UNIX"`. The second element is the path associated to the socket.

`path`

Returns the path associated to the socket.

`peeraddr`

Returns the array contains the peer's socket information. The elements of the array is same as the ones which `addr` returns.

`recvfrom(`*`len`*`[, `*`flags`*`])`

Receives data from the socket and returns the pair of data and the path of the sender. For arguments,

see `recv`.

UNIXServer

The server side of the UNIX stream socket.

SuperClass:

`UNIXSocket`

Methods:

`accept`

Accepts the request for the connection, and returns the `UNIXSocket` connected to the client.

Socket

`Socket` provides the low level access to the socket features of the operating system. Its methods are about same level of the Perl's socket functions. The socket addresses are represented by the C structure packed into the string.

Normally, the socket programming are done by the high level socket classes like `TCPSocket` and `TCPServer`.

SuperClass:

`BasicSocket`

Class Methods:

`open(`*`domain`*`, `*`type`*`, `*`protocol`*`)`
`new(`*`domain`*`, `*`type`*`, `*`protocol`*`)`

Creates new socket. *domain*, *type*, and *protocol* are specified by the constant found in the C header files. Most of the constants are defined as class constants in `Socket` class. *domain* and *type* can be specified by the string name. But all possible values may not available by the string.

`for_fd(`*`fd`*`)`

Creates new socket object corresponding the file discriptor *fd*.

`pair(`*`domain`*`, `*`type`*`, `*`protocol`*`)`
`socketpair(`*`domain`*`, `*`type`*`, `*`protocol`*`)`

Returns the pair of the connected sockets. See `socketpair(2)`. The argument specification is same to `Socket.open`.

`gethostbyname(`*`host`*`)`

Returns the array conaining the host information from the host name or the IP address (32 bit integer

or string such as `"127.0.0.1"`). The first element of the array is the hostname. The second element is the array of the host aliases (possibly empty). The third element is the address type. And sequence of the addresses follows. The addresses are packed string.

`gethostbyaddr(`*`host`*`)`

Returns the array conaining the host information from the packed `struct sockaddr`. Data in the array is as decribed in gethostbyname. host name or the IP address (32 bit integer or string such as `"127.0.0.1"`).

`getservbyname(`*`service`*`[, `*`proto`*`])`

Returns the port number corrensponding *service* and *proto*. The default value for the *proto* is `"tcp"`.

Methods:

`accept`

Accepts the connection and returns the pair of the socket for the new connection and address of the connection. See `accept(2)`.

`bind(`*`addr`*`)`

Binds the socket to the *addr*. See `bind(2)`. The *addr* is the sockaddr structure packed into the string.

`connect(`*`addr`*`)`

Connects the socket to the *addr*. The *addr* is the sockaddr structure packed into the string.

`listen(`*`backlog`*`)`

Specifies the connection queue limit for the socket. See `listen(2)`.

`recvfrom(`*`len`*`[, `*`flags`*`])`

Receives data from the socket and returns the pair of data and the address of the sender. For arguments, see `recv`.

The Thread Library

The `Thread` library allows concurrent programming in Ruby. It provides multiple threads of control that execute concurrently sharing same memory space. The Ruby interpreter executes threads by time-sharing, therefore using threads never makes program run faster (even much slower for the cost of context switching).

The thread which created first when the Ruby process started, is called the main thread. When the main thread is terminated for some reason, all other threads will be terminated, and the whole process will be terminated. The exception raised by the user interrupt is delivered to the main thread.

The `Thread` library is optional, and may not be available on some Ruby interpreter, which thread feature is disabled by the compile time configuration.

Thread

The `Thread` class represents user-level threads.

Threads terminates when the execution of the given iterater block returns, normally or by raisung an exception.

SuperClass:

Object

Class Methods:

`abort_on_exception`

Returns the value of the abort flag.

`abort_on_exception=(`*`yes_no`*`)`

Sets the value of the flag to abort the interpreter if any unhandled exception terminates any thread.

`current`

Returns the current thread object which calls the method.

`exit`

Terminates the current thread.

`kill(`*`thread`*`)`

Terminates the specified *thread*.

```
new {...}
start {...}
fork {...}
```

Creates a new thread of contorl, then starts evaluating the block concurrently. Returns a newly created thread object.

`pass`

Gives other runnable threads chance to run.

`stop`

Suspends the current thread until another thread resumes it using `run` method.

Methods:

`self[`*`name`*`]`

Retrieves thread local data associated with *name*. The *name* is either string or symbol.

`self[`*`name`*`]=`*`val`*

Stores *val* into the thread local dictionary, associating with *name*. The *name* is either string or symbol.

`abort_on_exception`

Returns the value of the abort flag of the thread.

`abort_on_exception=(`*`yes_no`*`)`

Sets the value of the flag to abort the interpreter if any unhandled exception terminates the thread.

`alive?`
`status`

Returns `true` if the thread is alive. Returns `false` for normal termination, `nil` for termination by exception.

`exit`

Terminates the thread.

`join`

Suspends the current thread until the thread has terminated.

`raise([`*`error_type`*`,][`*`message`*`][,`*`traceback`*`])`

Raises an exception on the thread.

`run`

Resumes the thread. It does nothing if the thread was not suspended.

`stop?`

Returns true if the thread is stopped.

`value`

Waits for the thread to terminate and returns the evaluated value of the block, which is given to the `Thread.create`.

`wakeup`

Resumes the thread.

Mutex

Mutexs (mutual-exclusion locks) are used to protect shared data against concurrent accesses. The typical use is (where `m` is the mutex object):

```
begin
  m.lock
  # critical section protected by m
ensure
  m.unlock
end
```

or, in short

```
m.synchronize {
  # critical section protected by m
}
```

SuperClass:

Object

Class Methods:

`new`

Creates a new `Mutex` object.

Methods:

`lock`

Locks the mutex object. Only on thread at a time can lock the mutex. A thread that attempts to lock a alread locked mutex will suspend until the other thread unlocks the mutex.

`locked?`

Returns true if the mutex is locked.

`synchronize`

Locks the mutex, and evaluates the block. Ensures the mutex be unlocked.

`try_lock`

Locks the mutex and returns true if the mutex is not locked. If the mutex is already locked, just returns false.

`unlock`

Unlocks the mutex. Other thread suspended trying to lock the mutex will be resumed.

Queue

The `Queue` is the FIFO (first in first out) communication channel between threads. If a thread trys to read an empty queue, it will be suspended until the queue is filled.

SuperClass:

Object

Class Methods:

`new`

Creates a new queue object.

Methods:

`empty?`

Returns true if the queue is empty.

`length`

Returns the length of the queue.

`pop [`*`non_block`*`]`

Removes and returns an value from queue. If the queue is empty, the calling thread will be suspended until some value pushed in the queue. If optional argument *non_block* is non-nil, `pop` raises an exception if no value available in the queue.

`push(`*`value`*`)`

Append an *value* to the queue. Restart the waiting thread if any.

Appendix A

Pseudo BNF Syntax of Ruby

Here is the syntax of Ruby in pseudo BNF. For more detail, see parse.y in Ruby distribution.

```
PROGRAM           : COMPSTMT

COMPSTMT : STMT (TERM EXPR)* [TERM]

STMT              : CALL do [`|' [BLOCK_VAR] `|'] COMPSTMT end
                | undef FNAME
                  | alias FNAME FNAME
                  | STMT if EXPR
                  | STMT while EXPR
                  | STMT unless EXPR
                  | STMT until EXPR
                | `BEGIN' `{' COMPSTMT `}'
                | `END' `{' COMPSTMT `}'
                | LHS `=' COMMAND [do [`|' [BLOCK_VAR] `|'] COMPSTMT end]
                  | EXPR

EXPR              : MLHS `=' MRHS
                  | return CALL_ARGS
                  | yield CALL_ARGS
                  | EXPR and EXPR
                  | EXPR or EXPR
                  | not EXPR
                  | COMMAND
                  | `!' COMMAND
                  | ARG

CALL              : FUNCTION
                | COMMAND

COMMAND           : OPERATION CALL_ARGS
                  | PRIMARY `.' OPERATION CALL_ARGS
                  | PRIMARY `::' OPERATION CALL_ARGS
```

```
                  | super CALL_ARGS

FUNCTION        : OPERATION [`(' [CALL_ARGS] `)']
                  | PRIMARY `.' OPERATION `(' [CALL_ARGS] `)'
                  | PRIMARY `::' OPERATION `(' [CALL_ARGS] `)'
                  | PRIMARY `.' OPERATION
                  | PRIMARY `::' OPERATION
                  | super `(' [CALL_ARGS] `)'
                  | super

ARG               : LHS `=' ARG
                  | LHS OP_ASGN ARG
                  | ARG `..' ARG
                  | ARG `...' ARG
                  | ARG `+' ARG
                  | ARG `-' ARG
                  | ARG `*' ARG
                  | ARG `/' ARG
                  | ARG `%' ARG
                  | ARG `**' ARG
                  | `+' ARG
                  | `-' ARG
                  | ARG `|' ARG
                  | ARG `^' ARG
                  | ARG `&' ARG
                  | ARG `<=>' ARG
                  | ARG `>' ARG
                  | ARG `>=' ARG
                  | ARG `<' ARG
                  | ARG `<=' ARG
                  | ARG `==' ARG
                  | ARG `===' ARG
                  | ARG `!=' ARG
                  | ARG `=~' ARG
                  | ARG `!~' ARG
                  | `!' ARG
                  | `~' ARG
                  | ARG `<<' ARG
                  | ARG `>>' ARG
                  | ARG `&&' ARG
                  | ARG `||' ARG
                  | defined? ARG
                  | PRIMARY

PRIMARY           : `(' COMPSTMT `)'
                  | LITERAL
                  | VARIABLE
                  | PRIMARY `::' IDENTIFIER
                  | `::' IDENTIFIER
                  | PRIMARY `[' [ARGS] `]'
                  | `[' [ARGS [`,']] `]'
                  | `{' [(ARGS|ASSOCS) [`,']] `}'
                  | return [`(' [CALL_ARGS] `)']
                  | yield [`(' [CALL_ARGS] `)']
                  | defined? `(' ARG `)'
                | FUNCTION
                  | FUNCTION `{' [`|' [BLOCK_VAR] `|'] COMPSTMT `}'
                  | if EXPR THEN
                    COMPSTMT
                    (elsif EXPR THEN COMPSTMT)*
                    [else COMPSTMT]
                    end
                  | unless EXPR THEN
                    COMPSTMT
                    [else COMPSTMT]
                    end
                  | while EXPR DO COMPSTMT end
                  | until EXPR DO COMPSTMT end
                  | case COMPSTMT
                    (when WHEN_ARGS THEN COMPSTMT)+
                    [else COMPSTMT]
                    end
                  | for BLOCK_VAR in EXPR DO
                    COMPSTMT
                    end
                  | begin
```

```
                    COMPSTMT
                    [rescue [ARGS] DO COMPSTMT]+
                    [else COMPSTMT]
                    [ensure COMPSTMT]
                    end
                  | class IDENTIFIER [`<' IDENTIFIER]
                    COMPSTMT
                    end
                  | module IDENTIFIER
                    COMPSTMT
                    end
                  | def FNAME ARGDECL
                    COMPSTMT
                    end
                  | def SINGLETON (`.'|`::') FNAME ARGDECL
                    COMPSTMT
                    end

WHEN_ARGS         : ARGS [`,' `*' ARG]
                  | `*' ARG

THEN              : TERM
                  | then
                  | TERM then

DO                : TERM
                  | do
                  | TERM do

BLOCK_VAR         : LHS
                  | MLHS

MLHS              : MLHS_ITEM `,' [MLHS_ITEM (`,' MLHS_ITEM)*] [`*' [LHS]]
                | `*' LHS

MLHS_ITEM         : LHS
                  | '(' MLHS ')'

LHS               : VARIABLE
                  | PRIMARY `[' [ARGS] `]'
                  | PRIMARY `.' IDENTIFIER

MRHS              : ARGS [`,' `*' ARG]
                  | `*' ARG

CALL_ARGS         : ARGS
                  | ARGS [`,' ASSOCS] [`,' `*' ARG] [`,' `&' ARG]
                  | ASSOCS [`,' `*' ARG] [`,' `&' ARG]
                  | `*' ARG [`,' `&' ARG]
                  | `&' ARG
                  | COMMAND

ARGS              : ARG (`,' ARG)*

ARGDECL           : `(' ARGLIST `)'
                  | ARGLIST TERM

ARGLIST           : IDENTIFIER(`,'IDENTIFIER)*[`,'`*'[IDENTIFIER]][`,'`&'IDENTIFIER]
                  | `*'IDENTIFIER[`,'`&'IDENTIFIER]
                  | [`&'IDENTIFIER]

SINGLETON         : VARIABLE
                  | `(' EXPR `)'

ASSOCS            : ASSOC (`,' ASSOC)*

ASSOC             : ARG `=>' ARG

VARIABLE : VARNAME
                  | nil
                  | self

LITERAL           : numeric
                  | SYMBOL
                  | STRING
                  | STRING2
```

```
                  | HERE_DOC
                  | REGEXP

TERM              : `;'
                  | `\n'
```

The followings are recognized by lexical analizer.

```
OP_ASGN           : `+=' | `-=' | `*=' | `/=' | `%=' | `**='
                  | `&=' | `|=' | `^=' | `<<=' | `>>='
                  | `&&=' | `||='

SYMBOL            : `:'FNAME
                  | `:'VARNAME

FNAME             : IDENTIFIER | `..' | `|' | `^' | `&'
                  | `<=>' | `==' | `===' | `=~'
                | `>' | `>=' | `<' | `<='
                  | `+' | `-' | `*' | `/' | `%' | `**'
                  | `<<' | `>>' | `~'
                | `+@' | `-@' | `[]' | `[]='

OPERATION         : IDENTIFIER
                  | IDENTIFIER'!'
                  | IDENTIFIER'?'

VARNAME           : GLOBAL
                  | `@'IDENTIFIER
                  | IDENTIFIER

GLOBAL            : `$'IDENTIFIER
                  | `$'any_char
                  | `$''-'any_char

STRING            : `"' any_char* `"'
                  | `'' any_char* `''
                  | `` any_char* ``'

STRING2           : `%'(`Q'|`q'|`x')char any_char* char

HERE_DOC          : `<<'(IDENTIFIER|STRING)
                    any_char*
                    IDENTIFIER

REGEXP            : `/' any_char* `/'[`i'|`o'|`p']
                  | `%'`r' char any_char* char

IDENTIFIER is the sqeunce of characters in the pattern of /[a-zA-Z_][a-zA-Z0-9_]*/.
```

Ruby Glossary

A

B

C

D

document

The one that matz is really bad at. He always says ``the source itself should be the document. It even describes bugs perfectly." But no one agrees.

E

F

G

H

I

J

Japanese (language)

Native language of matz, the author of Ruby. The cause of the poor documentation of Ruby (according to his excuse).

K

L

M

matz

Ruby's author. His real name is `Yukihiro Matsumoto'. If you say `*You-Key-Hero Matz-motor*' real quick, it sounds like his name. (`tz' as in `waltz')

N

O

P

Perl

Hmm.. Sounds familiar. What's that?

Q

R

Ruby

The name of the object-oriented script language and its interpreter. The name `ruby' is not an acronym. It's named after the red precious stone, which is the birthsone of July. Notice pearl(Perl) is the birthstone of

June.

S

T

U

V

W

X

Y

Z