Module
Kernel

The Kernel module is included by class Object, so its methods are available in every Ruby object. The Kernel instance methods are documented in class Object beginning on page 622. This section documents the private methods. These methods are called without a receiver and thus can be called in functional form.

Module methods

__callee__

__callee__ → *symbol* or nil

1.9 Returns the name of the current method or nil outside the context of a method.

```
def fred
  puts "I'm in #{__callee__.inspect}"
end
fred
puts "Then in #{__callee__.inspect}"
```

produces:

```
I'm in :fred
Then in nil
```

__method__

__method__ → *symbol* or nil

1.9 Synonym for __callee__.

Array

Array(*arg*) → *array*

1.9 Returns *arg* as an Array. First tries to call *arg*.to_ary, then *arg*.to_a. If both fail, creates a single element array containing *arg* (or an empty array if *arg* is nil).

```
Array(1..5)   # =>  [1, 2, 3, 4, 5]
```

Complex

Complex(*real*, *imag*=0) → *complex*

1.9 Returns the complex number with the given real and imaginary parts.

```
Complex(1)            # =>  1+0i
Complex("1")          # =>  1+0i
Complex("1", "3/2")   # =>  1+3/2i
Complex("3+2i")       # =>  3+2i
```

Float

Float(*arg*) → *float*

Returns *arg* converted to a float. Numeric types are converted directly; the rest are converted using *arg*.to_f. Converting nil generates a TypeError.

```
Float(1)           # =>  1.0
Float("123.456")   # =>  123.456
```

Integer

Integer(*arg*) → *int*

Converts *arg* to a Fixnum or Bignum. Numeric types are converted directly (floating-point numbers are truncated). If *arg* is a String, leading radix indicators (0, 0b, and 0x) are honored. Others are converted using to_int and to_i. This behavior is different from that of String#to_i. Converting nil generates a TypeError. 1.9

```
Integer(123.999)    # =>   123
Integer("0x1a")     # =>   26
Integer(Time.new)   # =>   1239647202
```

Rational

Rational(*numerator*, *denominator*=1) → *complex*

1.9 Returns the rational number with the given representation.

```
Rational(1)             # =>   1/1
Rational("1")           # =>   1/1
Rational("1", "2")      # =>   1/2
Rational(1, 0.5)        # =>   2/1
Rational("3/2")         # =>   3/2
Rational("3/2", "4/5")  # =>   15/8
```

String

String(*arg*) → *string*

Converts *arg* to a String by calling its to_s method.

```
String(self)        # =>   "main"
String(self.class)  # =>   "Object"
String(123456)      # =>   "123456"
```

` (backquote)

\`*cmd*\` → *string*

Returns the standard output of running *cmd* in a subshell. The built-in syntax %x{. . .} described on page 149 uses this method. Sets $? to the process status.

```
`date`                  # =>   "Mon Apr 13 13:26:42 CDT 2009\n"
`ls testdir`.split[1]   # =>   "main.rb"
`echo oops && exit 99`  # =>   "oops\n"
$?.exitstatus           # =>   99
```

abort

abort
abort(*msg*)

Terminates execution immediately with an exit code of 1. The optional String parameter is written to standard error before the program terminates.

at_exit

at_exit { *block* } → *proc*

Converts *block* to a Proc object (and therefore binds it at the point of call) and registers it for execution when the program exits. If multiple handlers are registered, they are executed in reverse order of registration.

```
def do_at_exit(str1)
  at_exit { print str1 }
end
at_exit { puts "cruel world" }
do_at_exit("goodbye ")
exit
```

produces:

```
goodbye cruel world
```

autoload

autoload(*name*, *file_name*) → nil

Registers *file_name* to be loaded (using Kernel.require) the first time that the module *name* (which may be a String or a symbol) is accessed.

```
autoload(:MyModule, "/usr/local/lib/modules/my_module.rb")
```

Module.autoload lets you define namespace-specific autoload hooks:

```
module X
 autoload :XXX, "xxx.rb"
end
```

Note that xxx.rb should define a class in the correct namespace. That is, in this example xxx.rb should contain the following:

```
class X::XXX
  # ...
end
```

autoload?

autoload?(*name*) → *file_name* or nil

Returns the name of the file that will be autoloaded when the string or symbol *name* is referenced in the top-level context or returns nil if there is no associated autoload.

```
autoload(:Fred, "module_fred")  # =>  nil
autoload?(:Fred)                # =>  "module_fred"
autoload?(:Wilma)               # =>  nil
```

binding

binding → *a_binding*

Returns a Binding object, describing the variable and method bindings at the point of call. This object can be used when calling eval to execute the evaluated command in this environment. Also see the description of class Binding beginning on page 469.

```
def get_binding(param)
  return binding
end
b = get_binding("hello")
eval("param", b)   # =>  "hello"
```

block_given?

block_given? → true or false

Returns true if yield would execute a block in the current context.

```
def try
  if block_given?
    yield
  else
    "no block"
  end
end
try                 # =>   "no block"
try { "hello" }     # =>   "hello"
block = lambda { "proc object" }
try(&block)         # =>   "proc object"
```

caller caller(⟨ *int* ⟩) → *array*

Returns the current execution stack—an array containing strings in the form *file:line* or *file:line: in 'method'*. The optional *int* parameter determines the number of initial stack entries to omit from the result.

```
def a(skip)
  caller(skip)
end
def b(skip)
  a(skip)
end
def c(skip)
  b(skip)
end
c(0)   # =>   ["/tmp/prog.rb:2:in `a'", "/tmp/prog.rb:5:in `b'",
              "/tmp/prog.rb:8:in `c'", "/tmp/prog.rb:10:in `<main>'"]
c(1)   # =>   ["/tmp/prog.rb:5:in `b'", "/tmp/prog.rb:8:in `c'",
              "/tmp/prog.rb:11:in `<main>'"]
c(2)   # =>   ["/tmp/prog.rb:8:in `c'", "/tmp/prog.rb:12:in `<main>'"]
c(3)   # =>   ["/tmp/prog.rb:13:in `<main>'"]
```

catch catch(*object*=Object.new) { *block* } → obj

1.9 catch executes its block. If a throw is encountered, Ruby searches up its stack for a catch block with a parameter identical to the throw's parameter. If found, that block is terminated, and catch returns the value given as the second parameter to throw. If throw is not called, the block terminates normally, and the value of catch is the value of the last expression evaluated. catch expressions may be nested, and the throw call need not be in lexical scope.
1.9 Prior to Ruby 1.9 the parameters to catch and throw had to be symbols—they can now be any object. When using literals, it probably makes sense to use only immediate objects.

```
def routine(n)
  print n, ' '
  throw :done if n <= 0
  routine(n-1)
end
catch(:done) { routine(4) }
```

produces:

```
4 3 2 1 0
```

chomp chomp(⟨ *rs* ⟩) → $_ or *string*

1.9 Equivalent to $_ = $_.chomp(*rs*), except no assignment is made if chomp doesn't change $_. See String#chomp on page 675. Available only with the -n or -p command-line options are present.

chop chop → *string*

1.9 (Almost) equivalent to ($_.dup).chop!, except that if chop would perform no action, $_ is unchanged and nil is not returned. See String#chop! on page 676. Available only with the -n or -p command-line options are present.

eval eval(*string* ⟨ , *binding* ⟨ , *file* ⟨ , *line* ⟩ ⟩ ⟩) → *obj*

1.9 Evaluates the Ruby expression(s) in *string*. If *binding* is given, the evaluation is performed in its context. The binding must be a Binding object. If the optional *file* and *line* parameters are present, they will be used when reporting syntax errors.

```
def get_binding(str)
  return binding
end
str = "hello"
eval "str + ' Fred'"                      # =>   "hello Fred"
eval "str + ' Fred'", get_binding("bye")  # =>   "bye Fred"
```

Local variables assigned within an eval are available after the eval only if they were defined at the outer scope before the eval executed. In this way, eval has the same scoping rules as blocks.

```
a = 1
eval "a = 98; b = 99"
puts a
puts b
```

produces:

```
98
prog.rb:4:in `<main>': undefined local variable or method `b' for
 main:Object (NameError)
```

exec exec(⟨ env, ⟩ *command* ⟨ , *args* ⟩*, ⟨ options ⟩)

1.9 Replaces the current process by running the given external command. If exec is given a single argument, that argument is taken as a line that is subject to shell expansion before being executed. If *command* contains a newline or any of the characters `*?{}[]<>()~\&|\$;'`"`, or under Windows if *command* looks like a shell-internal command (for example `dir`), *command* is run under a shell. On Unix system, Ruby does this by prepending `sh -c`. Under Windows, it uses the name of a shell in either RUBYSHELL or COMSPEC.

If multiple arguments are given, the second and subsequent arguments are passed as parameters to *command* with no shell expansion. If the first argument is a two-element array, the first element is the command to be executed, and the second argument is used as the argv[0] value, which may show up in process listings. In MSDOS environments, the command is executed in a subshell; otherwise, one of the exec(2) system calls is used, so the running command may inherit some of the environment of the original program (including open file descriptors). Raises SystemCallError if the *command* couldn't execute (typically Errno::ENOENT).

```
exec "echo *"       # echoes list of files in current directory
# never get here
exec "echo", "*"    # echoes an asterisk
# never get here
```

env, if present, is a hash that adds to the environment variables in the subshell. An entry with a nil value clears the corresponding environment variable. The keys must be strings. *options*, if present, is a hash that controls the setup of the subshell. The possible keys and their meanings are listed in Table 27.8 on page 580. See also Kernel.spawn and Kernel.system.

exit exit(true | false | *status*=1)

Initiates the termination of the Ruby script. If called in the scope of an exception handler, raises a SystemExit exception. This exception may be caught. Otherwise, exits the process using exit(2). The optional parameter is used to return a status code to the invoking environment. With an argument of true, exits with a status of zero. With an argument that is false (or no argument), exits with a status of 1; otherwise, exits with the given status. The default exit value is 1.

```
fork { exit 99 }
Process.wait
puts "Child exits with status: #{$?.exitstatus}"
 begin
   exit
   puts "never get here"
 rescue SystemExit
   puts "rescued a SystemExit exception"
 end
 puts "after begin block"
```

produces:

```
Child exits with status: 99
rescued a SystemExit exception
after begin block
```

Just prior to termination, Ruby executes any at_exit functions and runs any object finalizers (see ObjectSpace beginning on page 635).

```
at_exit { puts "at_exit function" }
ObjectSpace.define_finalizer("xxx",  lambda { |obj| puts "in finalizer" })
exit
```

produces:

```
at_exit function
in finalizer
```

exit!

exit!(true | false | *status*=1)

Similar to Kernel.exit, but exception handling, at_exit functions, and finalizers are bypassed.

fail

fail
fail(*message*)
fail(*exception* ⟨ , *message* ⟨ , *array* ⟩ ⟩)

Synonym for Kernel.raise.

fork

fork ⟨ { *block* } ⟩ → *int* or nil

Creates a subprocess. If a block is specified, that block is run in the subprocess, and the subprocess terminates with a status of zero. Otherwise, the fork call returns twice, once in the parent, returning the process ID of the child, and once in the child, returning nil. The child process can exit using Kernel.exit! to avoid running any at_exit functions. The parent process should use Process.wait to collect the termination statuses of its children or use Process.detach to register disinterest in their status; otherwise, the operating system may accumulate zombie processes.

```
fork do
  3.times {|i| puts "Child: #{i}" }
end
3.times {|i| puts "Parent: #{i}" }
Process.wait
```

produces:

```
Parent: 0
Child: 0
Child: 1
Child: 2
Parent: 1
Parent: 2
```

format

format(*format_string* ⟨ , *arg* ⟩*) → *string*

Synonym for Kernel.sprintf.

gem

1.9

gem(*gem_name* ⟨ , *version* ⟩) → true or false

Adds the given gem to the applications include path, so that subsequent requires will search. Defaults to the latest version of the gem if no version information is given. See section *Gems and Versions* on page 243 for more information and examples.

gets

gets(*separator*=$/) → *string* or nil

Returns (and assigns to $_) the next line from the list of files in ARGV (or $*) or from standard input if no files are present on the command line. Returns nil at end of file. The optional argument specifies the record separator. The separator is included with the contents of each record. A separator of nil reads the entire contents, and a zero-length separator reads the input one paragraph at a time, where paragraphs are divided by two consecutive newlines. If multiple filenames are present in ARGV, gets(nil) will read the contents one file at a time.

```
ARGV << "testfile"
print while gets
```

produces:

```
This is line one
This is line two
This is line three
And so on...
```

The style of programming using $_ as an implicit parameter is gradually losing favor in the Ruby community.

global_variables global_variables → *array*

Returns an array of the names of global variables.

```
global_variables.grep /std/   # =>   [:$stdin, :$stdout, :$stderr]
```

gsub gsub(*pattern*, *replacement*) → *string*
gsub(*pattern*) { *block* } → *string*

1.9 Equivalent to $_.gsub(...), except that $_ will be updated if substitution occurs. Available only with the -n or -p command-line options are present.

iterator? iterator? → true or false

Deprecated synonym for Kernel.block_given?.

lambda lambda { *block* } → *proc*

Creates a new procedure object from the given block. See page 364 for an explanation of the difference between procedure objects created using lambda and those created using Proc.new. Note that lambda is now preferred over proc.

```
prc = lambda { "hello" }
prc.call   # =>   "hello"
```

load load(*file_name*, *wrap*=false) → true

Loads and executes the Ruby program in the file *file_name*. If the filename does not resolve to an absolute path, the file is searched for in the library directories listed in $:. If the optional *wrap* parameter is true, the loaded script will be executed under an anonymous module, protecting the calling program's global namespace. In no circumstance will any local variables in the loaded file be propagated to the loading environment.

local_variables

local_variables → *array*

Returns the names of the current local variables.

```
fred = 1
for i in 1..10
   # ...
end
local_variables   # =>   [:fred, :i]
```

Note that local variables are associated with bindings.

```
def fred
  a = 1
  b = 2
  binding
end
freds_binding = fred
eval("local_variables", freds_binding)   # =>   [:a, :b]
```

loop

loop ⟨ { *block* } ⟩

Repeatedly executes the block.

```
loop do
  print "Type something: "
  line = gets
  break if line.nil?  || line =~ /^[qQ]/
  # ...
end
```

1.9 loop silently rescues the StopIteration exception, which works well with external iterators.

```
enum1 = [1, 2, 3].to_enum
enum2 = [10, 20].to_enum
loop do
  puts enum1.next + enum2.next
end
```

produces:

```
11
22
```

open

open(*name* ⟨ , *mode* ⟨ , *permission* ⟩ ⟩) → *io* or nil
open(*name* ⟨ , *mode* ⟨ , *permission* ⟩ ⟩) {| *io* | *block* } → *obj*

1.9 Creates an IO object connected to the given stream, file, or subprocess.

If *name* does not start with a pipe character (|), treats it as the name of a file to open using the specified mode defaulting to "r" (see the table of valid modes on page 547). If a file is being created, its initial permissions may be set using the third parameter, which is an integer. If this third parameter is present, the file will be opened using the low-level open(2) rather than fopen(3) call.

If a block is specified, it will be invoked with the IO object as a parameter, which will be automatically closed when the block terminates. The call returns the value of the block in this case.

If *name* starts with a pipe character, a subprocess is created, connected to the caller by a pair of pipes. The returned IO object may be used to write to the standard input and read from the standard output of this subprocess. If the command following the | is a single minus sign, Ruby forks, and this subprocess is connected to the parent. In the subprocess, the open call returns nil. If the command is not "–", the subprocess runs the command. If a block is associated with an open("|–") call, that block will be run twice—once in the parent and once in the child. The block parameter will be an IO object in the parent and nil in the child. The parent's IO object will be connected to the child's STDIN and STDOUT. The subprocess will be terminated at the end of the block.

```
open("testfile", "r:iso-8859-1") do |f|
  print f.gets
end
```

produces:

```
This is line one
```

Open a subprocess, and read its output:

```
cmd = open("|date")
print cmd.gets
cmd.close
```

produces:

```
Mon Apr 13 13:26:43 CDT 2009
```

Open a subprocess running the same Ruby program:

```
f = open("|-", "w+")
if f.nil?
  puts "in Child"
  exit
else
  puts "Got: #{f.gets}"
end
```

produces:

```
Got: in Child
```

Open a subprocess using a block to receive the I/O object:

```
open("|-") do |f|
  if f.nil?
    puts "in Child"
  else
    puts "Got: #{f.gets}"
  end
end
```

produces:

```
Got: in Child
```

p p(⟨ *obj* ⟩$^+$) → *obj*

1.9

For each object, writes *obj*.inspect followed by the current output record separator to the program's standard output. Also see the PrettyPrint library on page 790.

```
Info = Struct.new(:name, :state)
p Info['dave', 'TX']
```

produces:

```
#<struct Info name="dave", state="TX">
```

print print(⟨ *obj* ⟩*) → nil

Prints each object in turn to STDOUT. If the output field separator ($,) is not nil, its contents will appear between each field. If the output record separator ($\) is not nil, it will be appended to the output. If no arguments are given, prints $_. Objects that aren't strings will be converted by calling their to_s method.

```
print "cat", [1,2,3], 99, "\n"
$, = ", "
$\ = "\n"
print "cat", [1,2,3], 99
```

produces:

```
cat[1, 2, 3]99
cat, [1, 2, 3], 99,
```

printf printf(*io*, *format* ⟨ , *obj* ⟩*) → nil
printf(*format* ⟨ , *obj* ⟩*) → nil

Equivalent to

io.write sprintf(*format, obj* ...)

or

STDOUT.write sprintf(*format, obj* ...)

proc proc { *block* } → *a_proc*

Creates a new procedure object from the given block. Mildly deprecated in favor of Kernel#lambda.

```
prc = proc {|name| "Goodbye, #{name}" }
prc.call('Dave')   # =>   "Goodbye, Dave"
```

putc putc(*obj*) → *obj*

1.9

Equivalent to STDOUT.putc(*obj*). If *obj* is a string, output its first byte as a character; otherwise, attempts to convert *obj* to an integer and outputs the corresponding character code.

```
putc 65
putc 66.123
putc "CAT"
putc 12       # newline
```

produces:

```
ABC
```

puts puts(⟨ *arg* ⟩*) → nil

Equivalent to STDOUT.puts(*arg*...).

raise raise
raise(*message*)
raise(*exception* ⟨ , *message* ⟨ , *array* ⟩ ⟩)

1.9 With no arguments, raises the exception in $! or raises a RuntimeError if $! is nil. With a single String argument (or an argument that responds to to_str), raises a RuntimeError with the string as a message. Otherwise, the first parameter should be the name of an Exception class (or an object that returns an Exception when its exception method is called). The optional second parameter sets the message associated with the exception, and the third parameter is an array of callback information. Exceptions are caught by the rescue clause of begin. . . end blocks.

```
raise "Failed to create socket"
raise ArgumentError, "No parameters", caller
```

rand rand(*max*=0) → *number*

Converts *max* to an integer using $max_1 = max$.to_i.abs. If the result is zero, returns a pseudorandom floating-point number greater than or equal to 0.0 and less than 1.0. Otherwise, returns a pseudorandom integer greater than or equal to zero and less than max_1. Kernel.srand may be used to ensure repeatable sequences of random numbers between different runs of the program. Ruby currently uses a modified Mersenne Twister with a period of $2^{19937} - 1$.

```
srand 1234                  # =>   286573333591514665868880867940088004452
[ rand,  rand ]             # =>   [0.191519450378892, 0.622108771039832]
[ rand(10), rand(1000) ]    # =>   [4, 664]
srand 1234                  # =>   1234
[ rand,  rand ]             # =>   [0.191519450378892, 0.622108771039832]
```

readline readline(⟨ *separator*=$/ ⟩) → *string*

Equivalent to Kernel.gets, except readline raises EOFError at end of file.

readlines readlines(⟨ *separator*=$/ ⟩) → *array*

Returns an array containing the lines returned by calling Kernel.gets(*separator*) until the end of file.

require require(*library_name*) → true or false

Ruby tries to load *library_name*, returning true if successful. If the filename does not resolve to an absolute path, it will be searched for in the directories listed in $:. If the file has the extension .rb, it is loaded as a source file; if the extension is .so, .o, or .dll,[2] Ruby loads the shared library as a Ruby extension. Otherwise, Ruby tries adding .rb, .so, and so on, to

2. Or whatever the default shared library extension is on the current platform.

the name. The name of the loaded feature is added to the array in $". A feature will not be loaded if its name already appears in $".[3] require returns true if the feature was successfully loaded.

```
require 'my-library.rb'
require 'db-driver'
```

require_relative require_relative(*library_path*) → true or false

1.9

Requires a library whose path is relative to the file containing the call. Thus, if the file /usr/local/mylib/bin contains the file myprog.rb and that program contains the following line:

```
require_relative "../lib/mylib"
```

Ruby will look for mylib in /usr/local/mylib/lib.

require_relative cannot be called interactively in irb.

select select(*read_array* ⟨ , *write_array* ⟨ , *error_array* ⟨ , *timeout* ⟩ ⟩ ⟩) → *array* or nil

Performs a low-level select call, which waits for data to become available from input/output devices. The first three parameters are arrays of IO objects or nil. The last is a timeout in seconds, which should be an Integer or a Float. The call waits for data to become available for any of the IO objects in *read_array*, for buffers to have cleared sufficiently to enable writing to any of the devices in *write_array*, or for an error to occur on the devices in *error_array*. If one or more of these conditions are met, the call returns a three-element array containing arrays of the IO objects that were ready. Otherwise, if there is no change in status for *timeout* seconds, the call returns nil. If all parameters are nil, the current thread sleeps forever.

```
select( [STDIN], nil, nil, 1.5 )   # =>   [[#<IO:<STDIN>>], [], []]
```

set_trace_func set_trace_func(*proc*) → *proc*
set_trace_func(nil) → nil

Establishes *proc* as the handler for tracing or disables tracing if the parameter is nil. *proc* takes up to six parameters: an event name, a filename, a line number, an object ID, a binding, and the name of a class. *proc* is invoked whenever an event occurs. Events are c-call (calls a C-language routine), c-return (returns from a C-language routine), call (calls a Ruby method), class (starts a class or module definition), end (finishes a class or module definition), line (executes code on a new line), raise (raises an exception), and return (returns from a Ruby method). Tracing is disabled within the context of *proc*.

See the example starting on page 427 for more information.

sleep sleep(*numeric*=0) → *fixnum*

Suspends the current thread for *numeric* seconds (which may be a Float with fractional seconds). Returns the actual number of seconds slept (rounded), which may be less than that

3. As of Ruby 1.9 this name is converted to an absolute path, so that require 'a';require './a' will load a.rb just once.

asked for if the thread was interrupted by a SIGALRM or if another thread calls Thread#run. An argument of zero causes sleep to sleep forever.

```
Time.now    # =>   2009-04-13 13:26:43 -0500
sleep 1.9   # =>   2
Time.now    # =>   2009-04-13 13:26:45 -0500
```

spawn spawn(⟨ env, ⟩ *command* ⟨ , *args* ⟩*, ⟨ options ⟩) → *pid*

1.9 Executes *command* in a subshell, returning immediately. (Compare with Kernel.system, which waits for the command to complete before returning to the caller.) Returns the process ID for the subprocess running the command. The arguments are processed in the same way as for Kernel.exec on page 568. Raises SystemCallError if the *command* couldn't execute (typically `Errno::ENOENT`).

```
pid = spawn("echo hello")
puts "Back in main program"
rc, status = Process::waitpid2(pid)
puts "Status = #{status}"
```

produces:

```
Back in main program
hello
Status = pid 85719 exit 0
```

env, if present, is a hash that adds to the environment variables in the subshell. An entry with a nil value clears the corresponding environment variable. The keys must be strings.

```
pid = spawn({"FRED" => "caveman"}, "echo FRED = $FRED")
Process::waitpid2(pid)
```

produces:

```
FRED = caveman
```

options, if present, is a hash that controls the setup of the subshell. The possible keys and their meanings are listed in Table 27.8 on page 580.

```
reader, writer = IO.pipe
pid = spawn("echo '4*a(1)' | bc -l", [ STDERR, STDOUT ] => writer)
writer.close
Process::waitpid2(pid)
reader.gets   # =>   "3.14159265358979323844\n"
```

sprintf sprintf(*format_string* ⟨ , *arguments* ⟩*) → *string*

Returns the string resulting from applying *format_string* to any additional arguments. Within the format string, any characters other than format sequences are copied to the result.

1.9 A format sequence consists of a percent sign; followed by optional flags, width, and precision indicators, and an optional name; and then terminated with a field type character. The field type controls how the corresponding sprintf argument is to be interpreted, and the flags modify that interpretation. The flag characters are shown in Table 27.9 on page 581, and the field type characters are listed in Table 27.10.

The field width is an optional integer, followed optionally by a period and a precision. The width specifies the minimum number of characters that will be written to the result for this field. For numeric fields, the precision controls the number of decimal places displayed. As 1.9 of Ruby 1.9, number zero is converted to a zero-length string if a precision of 0 is given. For string fields, the precision determines the maximum number of characters to be copied from the string. (Thus, the format sequence %10.10s will always contribute exactly ten characters to the result.)

```
sprintf("%d %04x", 123, 123)               # =>   "123␣007b"
sprintf("%08b '%4s'", 123, 123)            # =>   "01111011␣'␣123'"
sprintf("%1$*2$s %2$d %1$s", "hello", 8)   # =>   "␣␣␣hello␣8␣hello"
sprintf("%1$*2$s %2$d", "hello", -8)       # =>   "hello␣␣␣␣-8"
sprintf("%+g:% g:%-g", 1.23, 1.23, 1.23)   # =>   "+1.23:␣1.23:1.23"
```

1.9 In Ruby 1.9, you can pass a hash as the second argument and insert values from this hash into the string. The notation <name> can be used between a percent sign and a field-type character, in which case the name will be used to look up a value in the hash, and that value will be formatted according to the field specification. The notation {name} is equivalent to <name>s, substituting the corresponding value as a string. You can use width and other flag characters between the opening percent sign and the {.

```
sprintf("%<number>d %04<number>x", number: 123)     # =>   "123␣007b"
sprintf("%08<number>b '%5{number}'", number: 123)   # =>   "01111011␣'␣␣123'"
sprintf("%6{k}: %{v}", k: "Dave", v: "Ruby")        # =>   "␣␣Dave:␣Ruby"
```

srand srand(⟨ *number* ⟩) → *old_seed*

1.9 Seeds the pseudorandom number generator to the value of *number*.to_i. If *number* is omitted or zero, seeds the generator using a system random number generator if available; otherwise, seeds it using a combination of the time, the process ID, and a sequence number. (This is also the behavior if Kernel.rand is called without previously calling srand, but without the sequence.) By setting the seed to a known value, scripts that use rand can be made deterministic during testing. The previous seed value is returned. Also see Kernel.rand on page 575.

sub sub(*pattern*, *replacement*) → $_
sub(*pattern*) { *block* } → $_

1.9 Equivalent to $_.sub(*args*), except that $_ will be updated if substitution occurs. Available only with the -n or -p command-line options are present.

syscall syscall(*fixnum* ⟨ , *args* ⟩*) → *int*

Calls the operating system function identified by *fixnum*. The arguments must be either String objects or Integer objects that fit within a native long. Up to nine parameters may be passed. The function identified by *fixnum* is system dependent. On some Unix systems, the numbers may be obtained from a header file called syscall.h.

```
syscall 4, 1, "hello\n", 6   # '4' is write(2) on our system
```

produces:

```
hello
```

system system(⟨ env, ⟩ *command* ⟨ , *args* ⟩*, ⟨ options ⟩) → true or false or nil

1.9

Executes *command* in a subshell, returning true if the command was found and ran successfully, false is the command exited with a nonzero exit status, and nil if the command failed to execute. An error status is available in $?. The arguments are processed in the same way as for Kernel.exec on page 568. *env*, if present, is a hash that adds to the environment variables in the subshell. An entry with a nil value clears the corresponding environment variable. The keys must be strings. *options*, if present, is a hash that controls the setup of the subshell. The possible keys and their meanings are listed in Table 27.8 on the following page. See also Kernel.spawn.

```
system("echo *")
system("echo", "*")
system({"WILMA" => "shopper"}, "echo $WILMA")
```

produces:

```
config.h main.rb
*
shopper
```

test test(*cmd*, *file1* ⟨ , *file2* ⟩) → *obj*

Uses the integer *cmd* to perform various tests on *file1* (Table 27.11 on page 582) or on *file1* and *file2* (Table 27.12).

throw throw(*symbol* ⟨ , *obj* ⟩)

Transfers control to the end of the active catch block waiting for *symbol*. Raises NameError if there is no catch block for the symbol. The optional second parameter supplies a return value for the catch block, which otherwise defaults to nil. For examples, see Kernel.catch on page 567.

trace_var trace_var(*symbol*, *cmd*) → nil
trace_var(*symbol*) {| *val* | *block* } → nil

Controls tracing of assignments to global variables. The parameter *symbol* identifies the variable (as either a string name or a symbol identifier). *cmd* (which may be a string or a Proc object) or the block is executed whenever the variable is assigned and receives the variable's new value as a parameter. Only explicit assignments are traced. Also see Kernel.untrace_var.

```
trace_var :$dave, lambda {|v| puts "$dave is now '#{v}'" }
$dave = "hello"
$dave.sub!(/ello/, "i")
$dave += " Dave"
```

produces:

```
$dave is now 'hello'
$dave is now 'hi Dave'
```

trap trap(*signal*, *proc*) → *obj*
trap(*signal*) { *block* } → *obj*

See the Signal module on page 668.

untrace_var untrace_var(*symbol* ⟨ , *cmd* ⟩) → *array* or nil

Removes tracing for the specified command on the given global variable and returns nil. If no command is specified, removes all tracing for that variable and returns an array containing the commands actually removed.

warn warn *msg*

Writes the given message to STDERR (unless $VERBOSE is nil, perhaps because the -W0 command-line option was given).

```
warn "Danger, Will Robinson!"
```

produces:

```
Danger, Will Robinson!
```

Table 27.8. Options to Spawn and System

Option	**Effect on new process**
:pgroup => true | 0 | *int*	If true or 0, the new process will be a process group leader. Otherwise, the process will belong to group *int*.
:rlimit_*xxx* => val | [cur, max]	Sets a resource limit. See Process.getrlimit for information on the available limits.
:unsetenv_others => true	Clears all environment variables; then sets only those passed in the *env* parameter.
:chdir => *dir*	Changes to directory *dir* before running the process.
:umask => *int*	Specifies the umask for the process.
fd_desc => *stream*	Sets the process's standard input, output, or error to *stream*. See the description that follows this table for information.
:close_others => true | false	By default, all file descriptors apart from 0, 1, and 2 are closed. You can specify false to leave them open.
io_obj => :close	Explicitly closes the file descriptor corresponding to *io_obj* in the child process.

The *fd_desc* parameter identifies an I/O stream to be opened or assigned in the child process. It can be one of :in, STDIN, or 0 to represent standard input; :out, STDOUT, or 1 to represent standard output; or :err, STDERR, or 2 to represent standard error. It can also be an array containing one or more of these, in which case all fds in the array will be opened on the same stream.

The *stream* parameter can be the following:

- One of :in, STDIN, or 0 to represent the current standard input; :out, STDOUT, or 1 to represent the current standard output; or :err, STDERR, or 2 to represent the current standard error.
- A string representing the name of a file or device.
- An array. The first element is the name of a file or device, the optional second element is the mode, and the optional third element the permission. See the description of File#new on page 512 for details.

Table 27.9. sprintf Flag Characters

Flag	Applies To	Meaning
␣ (space)	bdEefGgiouXx	Leaves a space at the start of positive numbers.
digit$	all	Specifies the absolute argument number for this field. Absolute and relative argument numbers cannot both be used in a sprintf string.
#	beEfgGoxX	Uses an alternative format. For the conversions b, o, X, and x, prefixes the result with b, 0, 0X, 0x, respectively. For E, e, f, G, and g, forces a decimal point to be added, even if no digits follow. For G and g, does not remove trailing zeros.
+	bdEefGgiouXx	Adds a leading plus sign to positive numbers.
-	all	Left-justifies the result of this conversion.
0 (zero)	bdEefGgiouXx	Pads with zeros, not spaces.
*	all	Uses the next argument as the field width. If negative, left-justifies the result. If the asterisk is followed by a number and a dollar sign, uses the indicated argument as the width.

Table 27.10. sprintf Field Types

Field	Conversion
B	Converts argument as a binary number (0B0101 if # modifier used).
b	Converts argument as a binary number (0b0101 if # modifier used).
c	Argument is the numeric code for a single character.
d	Converts argument as a decimal number.
E	Equivalent to e but uses an uppercase *E* to indicate the exponent.
e	Converts floating point-argument into exponential notation with one digit before the decimal point. The precision determines the number of fractional digits (defaulting to six).
f	Converts floating-point argument as [␣-]ddd.ddd, where the precision determines the number of digits after the decimal point.
G	Equivalent to g but uses an uppercase *E* in exponent form.
g	Converts a floating-point number using exponential form if the exponent is less than -4 or greater than or equal to the precision, or in d.dddd form otherwise.
i	Identical to d.
o	Converts argument as an octal number.
p	The value of *argument.inspect*.
s	Argument is a string to be substituted. If the format sequence contains a precision, at most that many characters will be copied.
u	Treats argument as an unsigned decimal number.
X	Converts argument as a hexadecimal number using uppercase letters. Negative numbers will be displayed with two leading periods (representing an infinite string of leading FFs).
x	Converts argument as a hexadecimal number. Negative numbers will be displayed with two leading periods (representing an infinite string of leading FFs.)

Table 27.11. File Tests with a Single Argument

Flag	Description	Returns
?A	Last access time for *file1*	Time
?b	True if *file1* is a block device	true or false
?c	True if *file1* is a character device	true or false
?C	Last change time for *file1*	Time
?d	True if *file1* exists and is a directory	true or false
?e	True if *file1* exists	true or false
?f	True if *file1* exists and is a regular file	true or false
?g	True if *file1* has the setgid bit set (false under NT)	true or false
?G	True if *file1* exists and has a group ownership equal to the caller's group	true or false
?k	True if *file1* exists and has the sticky bit set	true or false
?l	True if *file1* exists and is a symbolic link	true or false
?M	Last modification time for *file1*	Time
?o	True if *file1* exists and is owned by the caller's effective UID	true or false
?O	True if *file1* exists and is owned by the caller's real UID	true or false
?p	True if *file1* exists and is a fifo	true or false
?r	True if *file1* is readable by the effective UID/GID of the caller	true or false
?R	True if *file1* is readable by the real UID/GID of the caller	true or false
?s	If *file1* has nonzero size, returns the size; otherwise, returns nil	Integer or nil
?S	True if *file1* exists and is a socket	true or false
?u	True if *file1* has the setuid bit set	true or false
?w	True if *file1* exists and is writable by the effective UID/ GID	true or false
?W	True if *file1* exists and is writable by the real UID/GID	true or false
?x	True if *file1* exists and is executable by the effective UID/GID	true or false
?X	True if *file1* exists and is executable by the real UID/GID	true or false
?z	True if *file1* exists and has a zero length	true or false

Table 27.12. File Tests with Two Arguments

Flag	Description
?-	True if *file1* is a hard link to *file2*
?=	True if the modification times of *file1* and *file2* are equal
?<	True if the modification time of *file1* is prior to that of *file2*
?>	True if the modification time of *file1* is after that of *file2*