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*[`, `*filetag*[`, `*lineno*]]]`)`

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.

When the *expr* contains nested methods, it is useful to have better traceback information than simply citing the `eval` and a line number. The *filetag* provides this in two ways. If it is left as the default (which is `"(eval)"`) then there is no traceback into the nested methods. If it is set to anything else then there is full traceback information, and also the tag in the error message is changed to this value.

The *lineno* defaults to 1, and is used as the starting line number of *expr* when producing any error messages.

`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(arg1...)`

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.

---

matz@netlab.co.jp