— Ruby Syntax —
A Reference Catalog of Expressions, Statements & Idioms

Ruby Language Keywords — The following are the "basic vocabulary" of Ruby. These keywords are *reserved* words, and cannot be redefined for any other purpose.

Unlike some other programming languages, Ruby has a very short list of keywords (only 40), and these handle primarily structural (blocks) and flow control (conditionals, looping) issues. The real "workhorses" of Ruby are found in its class and instance *methods*, in its Standard Library, Gems and application-specific classes, modules and methods (the ones you write).

Keywords

Alphabetical:

alias	else	next	then
and	elsif	nil	true
begin	end	not	undef
BEGIN [1]	END [1]	or	unless
break	ensure	redo	until
case	false	rescue	when
class	for	retry	while
def	if	return	yield
defined?	in	self	__FILE__ [1,2]
do	module	super	__LINE__ [1,2]

Categorical:

primitive values:	false, nil, true, self
blocks:	begin, BEGIN,[1] class, def, do, end, END,[1] module, undef
flow control:	break, case, else, elsif, if, next, redo, retry, return, super, then, unless, when, yield
conditionals:	and, in, not, or
loops:	for, until, while
exception handling:	ensure, rescue
miscellaneous:	alias, defined?, __FILE__, __LINE__ [1,2]

Notes:

[1] These keywords appear in all UPPERCASE: BEGIN, END, __FILE__ and __LINE__. BEGIN and END are completely distinct from begin and end.

[2] These keywords are spelled with two underscores "__" at their beginnings and ends: __FILE__ and __LINE__.

Expressions — In Ruby, anything that can reasonably return a value... does return a value; (just about) everything is an expression. Most things that are considered statements in other languages are expressions in Ruby.

= — Assignment ...is an expression!

variable **=** *expression*

examples:

```
x = 2 + 3          # => 5
x = y = z = 0      # => 0
```

obj.**method** — Invocation of a method: "sends a message to the object." A method invocation always returns a value.

examples:

```
'abc'.upcase       # => 'ABC'
s = "abc".upcase   # => 'ABC', assigned to obj-var s
a, b, c = 2, 3, 3  # => [2, 3, 3]
(a*b).**(c)        # => 6**3 => 216 – Yes, arithmetic operators
                   #                  are really methods, too!
```

? : — Ternary if-then-else

condition **?** *value-if-true* **:** *value-if-false*

examples:

```
x < 0 ? 0 : x.sqrt      # => root of positive x, otherwise 0;
y = x < 0 ? 0 : x.sqrt  # value of expr can be assigned
```

Booleans: &&, **||**, **!** — **and**, **or**, **not**

— **&&**, **||**, **and** and **or** use short-circuit evaluation

— *Important to know*: **nil** and **false** are "*false values*" (Boolean-wise); all other values, including **0** and **""**, are "*true values*"

examples:

```
nil && true      # => nil (which is false)
false and true   # => false
'foo' && 'boo'   # => 'boo' (which is true)
nil || true      # => true
false or true    # => true
'foo' || 'boo'   # => 'foo' (which is true)
!false           # => true
!nil             # => true
!true            # => false
!'boo'           # => false
not !'foo'       # => true
```

Comparisons — **<**, **<=**, **>=**, **>**

==, **===**, **!=**, **<=>**, **=~**, **!~**

examples:

```
1 < 2              # => true
1 <= 2             # => true
1 == 2             # => false
2 == 2             # => true
1 === 1            # => true
1 >= 2             # => false
1 > 2              # => false
1 != 2             # => true
1 <=> 2            # => -1
1 <=> 1            # =>  0
2 <=> 1            # =>  1
/\d\d/ =~ 'abcdef' # =>  1
/\d\d/ =~ 'x10abc' # =>  1
'10abc' =~ /\d\d/  # =>  0
/\d\d/ !~ '99abc'  # => false
/\d\d/ !~ 'abcde'  # => true
```

defined? — returns **nil** if its argument is undefined; if argument *is* defined, returns a description of that argument. Keyword, not a method.

defined? *argument*

examples:

```
defined? gronk         # => nil
defined? nil           # => "nil"
defined? 3.14          # => "expression"
defined? "test"        # => "expression"
defined? Math::E       # => "constant"
defined? x = 1         # => "assignment"
defined? 'str'.upcase  # => "method"
defined? puts          # => "method"
```

Class IO: Basic I/O — Input —

each — Executes its block for every "line" in io/file, lines terminated by separator, by default = '\n'

obj.**each { |** *itervar* **|** *statement(s)* **}**

or:

obj.**each do |** *itervar* **|**
statement(s)
end

example:

```
File.open( 'test.txt' ).each { | line | puts line.upcase }
File.open( 'test.txt' ).each do | line |
  puts line.upcase
end
```

gets — Reads a "line" returned as a string, lines terminated by separator, by default = '\n', returns **nil** at end-of-file

gets(*separator*)

example:

```
File.open( 'test.txt' ) do | f |
  while line = f.gets
    puts line.chomp.upcase
  end
end
```

readline — Reads a "line" returned as a string, terminated by separator, by default = '\n', raises EOFError at end-of-file

readline(*separator*)

example:

```
File.open( 'test.txt' ) do | f |
  begin
    while line = f.readline
      puts line.upcase
    end
  rescue EOFError
    puts '>> End-of-file'
  end
end
```

See also: **each_byte**, **each_char**, **getbyte**, **getc**, **read**, **readbyte**, **readchar**, **readlines**, **readpartial**, **read_nonblock**

Related: **scanf**

Class IO: Basic I/O — Output —

puts — always appends "\n" *unless* a newline already terminates the output string

puts [*expression*]

examples:

```
puts "This is a test"  # => nil, outputs string to stdout
a, b, c = 1, 2, 3
puts a, b, c           # => nil, outputs each value on
1                      #         a separate line (\n)
2
3
stderr.puts "Error: Danger, Will Robinson!"
```

pp — To use pp, script must: **require 'pp'**

pp *object*

examples:

```
pp "This is a test"  # => the object, outputs a formatted
a, b, c = 1, 2, 3    #                 string to stdout
pp a, b, c           # => [1, 2, 3], outputs each object
1                    #               on a separate line (\n)
2
3
```

See also: **p**, **print**, **printf**, **write**, **write_nonblock**
Related: **sprintf**

Class IO: Basic I/O — Close & miscellaneous —

close — Closes an io object and flushes any pending writes to the operating system

obj.**close**

example:

```
f = File.open( 'test.txt' )
...
f.close
```

See also: **close_read**, **close_write**, **closed?**, **flush**, **fsync**, **sync**

Statements — Ruby programs (or scripts) are simply sequences of statements, executed in-order from first-to-last, top-to-bottom. Ruby prefers to talk about *expressions* rather than *statements*, but it seems to consider statements to be the kind-of compound(y) (multi-line) executable constructs, comprised of one or more expressions… Mostly, *expression* and *statement* tend to get used interchangeably.

block — A chunk of code enclosed within **do**...**end** or braces **{**...**}**, to delimit a unit of code which is usually iteratively or conditionally executed.

```
do                  # do...end are used for multi-line blocks
  statement(s)
end
```

or (commonly used):

```
{ statement }    # braces { } are usually used for one-line blocks
```

or (rarely seen):

```
{                   # braces { } are not used for multi-line blocks
  statement(s)
}
```

Frequently used with iterators — one or more "itervars" appear between the "goal-posts" |...| and are like parameters to the block of code:

```
object.iterator_method do | itervar [,...] |
  statement(s)
end
```

or:

```
object.iterator_method { | itervar [,...] | statement }
```

examples:

```
3.times { puts "knock-knock-knock... Penny!" }

x = 0
[1,2,3,4].each { | e | x += e }
puts x           # => nil, outputs 10

%w{ wolf bear lion eagle }.each do | scout |
  badge = scout.capitalize
  puts "Scout level: #{badge}"
end
```

case — Multi-way **if-elsif-else**...

```
case select_expression
when value_expression_1
  statement(s)
[when value_expression_2
  statement(s)]...
[when value_expression_N
  statement(s)]...
[else                       # the “default case”
  statement(s)]...
end
```

or:

```
case                        # note: no selector
when conditional_expression_1
  statement(s)
[when conditional_expression_2
  statement(s)]...
[when conditional_expression_N
  statement(s)]...
[else                       # the “default case”
  statement(s)]...
end
```

examples:

```
that_song = ‘As Time Goes By’   # compare this to if-elsif-else below
case
when customer.next == ‘Ilsa Lund’
  bogey.says “Of all the gin joints in all the towns” +
             “ in all the world, she walks into mine.”
  bergman.says “Play it once, Sam, for old times' sake.” +
               “ Play it, Sam. Play \“#{that_song}\”.”
when song.name == that_song
  bogey.says “You played it for her, you can play it for me!” +
             “ ...If she can stand it, I can! Play it!”
when movie.set == :airport && movie.atmosphere == :fog
  bogey.says “Louis, I think this is the beginning” +
             “ of a beautiful friendship.”
else
  piano.play( that_song )
end
```

class — Define or open a class

```
class Classname          # Class names are always capitalized
  statement(s)
end
```

examples:

```
class Music
  attr_reader :genre, :era
  def initialize( genre, era )
    @genre = genre
    @era   = era
  end
end  # class Music
```

def — Define a method

```
def methodname[( p1 [, p2]... )]  # Method names are never capitalized
  statement(s)
end
```

examples:

```
def askprompted( pstr, dstr = "Y" )
  # expects a Yes or No response,
  # returns true for any response beginning with "Y" or "y",
  # returns false for everything else...
  # but does test & respond to exit/quit/Ctrl-D/Ctrl-Z...
  default ||= dstr
  prompt = pstr + ( default == "" ? " (y/n)? " : " (y/n) [#{default}]? " )
  answer = readline( prompt, true ).strip.downcase
  exit true if answer == "exit" || answer == "quit"
  answer = default.downcase if answer == ""
  return ( answer[0] == "y" ? true : false )
rescue StandardError
  exit true  # this exit always provides cmd-line status:0
end
```

for – in — Ruby does not have a "classical" for-loop like C or Pascal or Basic; it's really syntactic sugar for an iterator…

This "syntactic sugar" translates into... this:	

for var **in** *enumeration*
  [*statement(s)...*]
end
	***enumeration*.each do** | *itervar* |
	  [*statement(s)...*]
	end

Advice: Experienced Ruby coders don't use for-loops; they use *iterators over enumerable objects* instead, as these are much more flexible and powerful.

if — Note: Ruby's **then** keyword is optional *unless* statement is on same line as **if**...

```
if conditional_expression then statement
if conditional_expression [ then ]
  statement(s)
end
```

examples:

```
if line[0] == ' '
  line.lstrip.downcase
end

if line[0] == ' ' then line.lstrip.downcase

fname = 'test.rb'
f = File.open( fname )
line = f.gets
if f.lineno == 1 && line[0..1] == "#!"
  puts "File #{fname} has a shebang line!..."
  line.shebang.process
end
```

if – else — Note: Ruby's **then** keyword is optional *unless* statement is on same line as **if**...

```
if conditional_expression [ then ]
  statement(s)
else
  statement(s)
end
```

examples:

```
if que.empty? then que.reload!
else
  que.process!
end

if que.empty?    # this format is preferred, without the then
  que.reload!
else
  que.process!
end
```

if – elsif – else — Notice the spelling of “elsif”...

```
if conditional_expression1 [ then ]
  statement(s)
elsif conditional_expression2
  statement(s)
[elsif conditional_expressionN
  statement(s)
]...
else
  statement(s)
end
```

examples:

```
that_song = ‘As Time Goes By’   # compare this to case above
if customer.next == ‘Ilsa Lund’
  bogey.says “Of all the gin joints in all the towns” +
             “ in all the world, she walks into mine.”
  bergman.says “Play it once, Sam, for old times' sake.” +
               “ Play it, Sam. Play \“#{that_song}\”.”
elsif song.name == that_song
  bogey.says “You played it for her, you can play it for me!” +
             “ ...If she can stand it, I can! Play it!”
elsif movie.set == :airport && movie.atmosphere == :fog
  bogey.says “Louis, I think this is the beginning” +
             “ of a beautiful friendship.”
else
  piano.play( that_song )
end
```

if statement-modifier — Conditionally execute a statement

```
statement if conditional_expression
```

examples:

```
puts “>> debugging data” if options[:debug]
x = 1.0 if x.nil?
cp( ARG[0], ‘~/scratch/’ ) if ARG[0]
Dir.entries.each do | f |
  next if f == ‘.’ or f == ‘..’
  process_file( f )
end
```

loop — Repeatedly executes its block (this is a Kernel method, not a keyword). Note that this is an "infinite loop" — there is no controlling conditional, so loop termination must be done manually with a break-test.

```
loop do
  break if condition
  statement(s)
end
```

examples:

```
loop do
  print "Continue processing, or wait (C/w)? "
  answer = gets    # blocks for terminal input here...
  break if answer.lstrip =~ /^[Cc]/
end

# If a loop uses an enumerator (one or more), the loop will
# terminate cleanly as soon as the enumerator runs dry...
abf = ['ack','bar','far','foo'].to_enum
rng = (1..8).to_enum
loop do
  puts "#{rng.next} – #{abf.next}"
end
```

generates:

```
1 – ack
2 – bar
3 – far
4 – foo
```

module — Define or open a module, creates *namespaces* and supports *mixins*

```
module Modulename          # Module names are always capitalized
  statement(s)
end
```

examples:

```
module Musical
  KEYSONPIANO = 88
  STRINGSONBASS = 4
  DRUMSINKIT = 5

  def jazz_trio
    return [ KEYSONPIANO, STRINGSONBASS, DRUMSINKIT ]
  end
end
```

return — Returns a value from a method

return [*expression*]

Note that the return-expression is optional; if it's missing, then the value of the *last expression executed* is returned.

examples:

```
def some_method( ... )
  # some calculation...
  ...
  return result
end  # some_method

def another_method( ... )
  # some calculation...
  ...
  return nil if error_happened
end  # another_method

def yet_another_method( ... )
  # some calculation...
  ...
  return count != 1 ? 's' : ''
end  # yet_another_method
```

unless – else — Inverse of **if-else**, but seldom used with its **else** clause

unless *conditional_expression* [**then**]
  statement(s)
[**else**
  statement(s)]
end

examples:

```
unless !que.empty?
  que.reload!
else
  que.process!
end

unless que.empty?
  que.process!
end
```

unless statement-modifier — Conditionally execute a statement

statement **unless** *conditional_expression*

examples:

```
puts ">> debugging data" unless options[:quiet]

x = 1.0 unless ! x.nil?

cp( ARG[0], '~/scratch/' ) unless ARG[0].nil? || ARG[0] == ""

Dir.entries.each do | f |
  process_file( f ) unless f == '.' or f == '..'
end
```

until — Loop until done

```
until condition_becomes_true
  statement(s)
end
```

example:

```
x = 10
epsilon = 0.00001
iteration = 1
until x <= epsilon
  puts “iteration #{iteration} - x = #{x}”
  x /= 2.0
  iteration += 1
end
```

until statement-modifier — Conditionally loop on a statement

```
statement until conditional_expression  # becomes true...
```

examples:

```
x += 5 until x > 25
```

while — Loop while not-yet done

```
while condition_is_true
  statement(s)
end
```

example:

```
f = File.open( ‘~/projects/test.txt’ )
while line = f.gets
  puts( line ) if line[0] != ‘#’  # uncomment the file!
end
```

while statement-modifier — Conditionally loop on a statement

```
statement while conditional_expression  # is true...
```

examples:

```
x += 5 while x <= 25
```

Idioms — Idioms[1] are Ruby's "slang," conventional ways of expressing commonly used structures and expressions.

||= — Initialize object if it is **nil**

```
myvar ||= value   # tests and sets obj-reference (variable),
                  # if myvar is nil, assignes value to it,
                  # otherwise leaves it unchanged
```

examples:

```
default ||= 'yes'    # => 'yes'
```

method? — Method names ending in '**?**' usually determine a Boolean value (confirm a state or condition, like "is this object nil?").

obj.**method?**

examples:

```
nil.nil?                    # => true
foo = 7
foo.nil?                    # => false
foo = nil
foo.nil?                    # => true
8.even?                     # => true
8.odd?                      # => false
''.empty?                   # => true
[0,1].empty?                # => false
"pqr".kind_of?( String )    # => true
[1,2].kind_of?( Integer )   # => false
...
loop do
  line = f.gets             # Read a line from a file...
  break if f.eof?           # Terminate loop if end-of-file
end
```

method! — Method names ending in '**!**' usually modify (alter, change) their receiver (object). (Some Ruby coders say that '**!**' signifies a "*dangerous*" method, but this is a misleading and wrong way to characterize these methods.)

obj.**method!**

examples:

```
obj = %w{ Carl Elsa Ally Dave Bret }
obj.sort!                   # => obj = ["Ally","Bret","Carl","Dave","Elsa"]
obj.reverse!                # => obj = ["Elsa","Dave","Carl","Bret","Ally"]
...
loop do
  line = f.gets             # Read a line from a file...
  line.chomp!               # Remove trailing newline '\n' from line
  break if f.eof?
end
```

1 Wikipedia: en.wikipedia.org/wiki/Idiom

local names — Local names are used for local variables, method names and method parameters. All local names begin with a *lowercase letter* '**a**'..'**z**' or (rarely) with an underscore '**_**'.

example:

```
def write_report( title, plength = 60, dfont = '' )
  font = 'Courier' unless dfont != ''
  ...
end
```

$Global names — Global variables always start with a *dollar sign* '**$**', and have scope visibility throughout the Ruby program (script); they have "global scope."

example:

```
$Global_state = :base
puts $0
puts $PROGRAM_NAME    # $0 and $PROGRAM_NAME are the same value
```

@Instance variables — Instance variables are the "state values" for a class instance, and always begin with a *single at-sign* '**@**'. An instance variable has scope only within its class instance (is invisible outside that class), belongs to that instance, and must be accessed through an attribute method.

example:

```
class Music
  attr_reader :genre, :era
  def initialize( genre, era )
    @genre = genre
    @era   = era
  end
end  # class Music
```

@@Class variables — Class variables are "global to the class," and always begin with *double at-signs* "**@@**". A class variable has scope only within its class (is invisible outside that class), belongs to the class itself, and are usable only by the class and its instances — class variables are rarely used.

example:

```
class Music
  @@Context = :Jazz
  ...
end  # class Music
```

Constants, Modules & Class Names — Names of constants, classes and modules each begin with an UpperCase Letter '**A**'..'**Z**'. Conventionally, a CONSTANT name is always completely UPPERCASE, while names of Classes and Modules are always MixedCase.

example:

```
module Arts
  AUDIENCE = :worldwide
  class Music
    AUDSURFACE = 4 * Math::PI * r**2
  end  # class Music
end  # module Arts
```