File: control_expressions.rdoc [Ruby 2.2.0]

Control Expressions¶ ↥

Ruby has a variety of ways to control execution. All the expressions described here return a value.

For the tests in these control expressions, `nil` and `false` are false-values and `true` and any other object are true-values. In this document “true” will mean “true-value” and “false” will mean “false-value”.

`if` Expression¶ ↥

The simplest `if` expression has two parts, a “test” expression and a “then” expression. If the “test” expression evaluates to a true then the “then” expression is evaluated.

Here is a simple if statement:

```
if true then
  puts "the test resulted in a true-value"
end
```

This will print “the test resulted in a true-value”.

The `then` is optional:

```
if true
  puts "the test resulted in a true-value"
end
```

This document will omit the optional `then` for all expressions as that is the most common usage of `if`.

You may also add an `else` expression. If the test does not evaluate to true the `else` expression will be executed:

```
if false
  puts "the test resulted in a true-value"
else
  puts "the test resulted in a false-value"
end
```

This will print “the test resulted in a false-value”.

You may add an arbitrary number of extra tests to an if expression using `elsif`. An `elsif` executes when all

tests above the `elsif` are false.

```
a = 1

if a == 0
  puts "a is zero"
elsif a == 1
  puts "a is one"
else
  puts "a is some other value"
end
```

This will print “a is one” as `1` is not equal to `0`. Since `else` is only executed when there are no matching conditions.

Once a condition matches, either the `if` condition or any `elsif` condition, the `if` expression is complete and no further tests will be performed.

Like an `if`, an `elsif` condition may be followed by a `then`.

In this example only “a is one” is printed:

```
a = 1

if a == 0
  puts "a is zero"
elsif a == 1
  puts "a is one"
elsif a >= 1
  puts "a is greater than or equal to one"
else
  puts "a is some other value"
end
```

The tests for `if` and `elsif` may have side-effects. The most common use of side-effect is to cache a value into a local variable:

```
if a = object.some_value
  # do something to a
end
```

The result value of an `if` expression is the last value executed in the expression.

Ternary if¶ ↥

You may also write a if-then-else expression using `?` and `:`. This ternary if:

```
input_type = gets =~ /hello/i ? "greeting" : "other"
```

Is the same as this `if` expression:

```
input_type =
  if gets =~ /hello/i
    "greeting"
  else
    "other"
  end
```

While the ternary if is much shorter to write than the more verbose form, for readability it is recommended that the ternary if is only used for simple conditionals. Also, avoid using multiple ternary conditions in the same expression as this can be confusing.

`unless` Expression¶ ↥

The `unless` expression is the opposite of the `if` expression. If the value is false the “then” expression is executed:

```
unless true
  puts "the value is a false-value"
end
```

This prints nothing as true is not a false-value.

You may use an optional `then` with `unless` just like `if`.

Note that the above `unless` expression is the same as:

```
if not true
  puts "the value is a false-value"
end
```

Like an `if` expression you may use an `else` condition with `unless`:

```
unless true
  puts "the value is false"
else
  puts "the value is true"
```

```
end
```

This prints “the value is true” from the `else` condition.

You may not use `elsif` with an `unless` expression.

The result value of an `unless` expression is the last value executed in the expression.

Modifier `if` and `unless`¶ ↥

`if` and `unless` can also be used to modify an expression. When used as a modifier the left-hand side is the "then" expression and the right-hand side is the "test" expression:

```
a = 0

a += 1 if a.zero?

p a
```

This will print 1.

```
a = 0

a += 1 unless a.zero?

p a
```

This will print 0.

While the modifier and standard versions have both a “test” expression and a “then” expression, they are not exact transformations of each other due to parse order. Here is an example that shows the difference:

```
p a if a = 0.zero?
```

This raises the NameError “undefined local variable or method `a'”.

When ruby parses this expression it first encounters `a` as a method call in the “then” expression, then later it sees the assignment to `a` in the “test” expression and marks `a` as a local variable.

When running this line it first executes the “test” expression, `a = 0.zero?`.

Since the test is true it executes the “then” expression, `p a`. Since the `a` in the body was recorded as a method which does not exist the NameError is raised.

The same is true for `unless`.

`case` Expression¶ ↥

The `case` expression can be used in two ways.

The most common way is to compare an object against multiple patterns. The patterns are matched using the +===+ method which is aliased to +==+ on Object. Other classes must override it to give meaningful behavior. See Module#=== and Regexp#=== for examples.

Here is an example of using `case` to compare a String against a pattern:

```
case "12345"
when /^1/
  puts "the string starts with one"
else
  puts "I don't know what the string starts with"
end
```

Here the string `"12345"` is compared with `/^1/` by calling `/^1/ === "12345"` which returns `true`. Like the `if` expression the first `when` that matches is executed and all other matches are ignored.

If no matches are found the `else` is executed.

The `else` and `then` are optional, this `case` expression gives the same result as the one above:

```
case "12345"
when /^1/
  puts "the string starts with one"
end
```

You may place multiple conditions on the same `when`:

```
case "2"
when /^1/, "2"
  puts "the string starts with one or is '2'"
end
```

Ruby will try each condition in turn, so first `/^1/ === "2"` returns `false`, then `"2" === "2"` returns `true`, so “the string starts with one or is ‘2’” is printed.

You may use `then` after the `when` condition. This is most frequently used to place the body of the `when` on a single line.

```
case a
when 1, 2 then puts "a is one or two
when 3    then puts "a is three"
else           puts "I don't know what a is"
end
```

The other way to use a `case` expression is like an if-elsif expression:

```
a = 2

case
when a == 1, a == 2
  puts "a is one or two"
when a == 3
  puts "a is three"
else
  puts "I don't know what a is"
end
```

Again, the `then` and `else` are optional.

The result value of a `case` expression is the last value executed in the expression.

`while` Loop¶ ↥

The `while` loop executes while a condition is true:

```
a = 0

while a < 10 do
  p a
  a += 1
end

p a
```

Prints the numbers 0 through 10. The condition `a < 10` is checked before the loop is entered, then the body executes, then the condition is checked again. When the condition results in false the loop is terminated.

The `do` keyword is optional. The following loop is equivalent to the loop above:

```
while a < 10
  p a
```

```
  a += 1
end
```

The result of a `while` loop is `nil` unless `break` is used to supply a value.

`until` Loop¶ ↥

The `until` loop executes while a condition is false:

```
a = 0

until a > 10 do
  p a
  a += 1
end

p a
```

This prints the numbers 0 through 11. Like a while loop the condition `a > 10` is checked when entering the loop and each time the loop body executes. If the condition is false the loop will continue to execute.

Like a `while` loop the `do` is optional.

Like a `while` loop the result of an `until` loop is nil unless `break` is used.

`for` Loop¶ ↥

The `for` loop consists of `for` followed by a variable to contain the iteration argument followed by `in` and the value to iterate over using each. The `do` is optional:

```
for value in [1, 2, 3] do
  puts value
end
```

Prints 1, 2 and 3.

Like `while` and `until`, the `do` is optional.

The `for` loop is similar to using each, but does not create a new variable scope.

The result value of a `for` loop is the value iterated over unless `break` is used.

The `for` loop is rarely used in modern ruby programs.

Modifier `while` and `until`¶ ↥

Like `if` and `unless`, `while` and `until` can be used as modifiers:

```
a = 0

a += 1 while a < 10

p a # prints 10
```

`until` used as a modifier:

```
a = 0

a += 1 until a > 10

p a # prints 11
```

You can use `begin` and `end` to create a `while` loop that runs the body once before the condition:

```
a = 0

begin
  a += 1
end while a < 10

p a # prints 10
```

If you don't use `rescue` or `ensure` Ruby optimizes away any exception handling overhead.

`break` Statement¶ ↥

Use `break` to leave a block early. This will stop iterating over the items in `values` if one of them is even:

```
values.each do |value|
  break if value.even?

  # ...
end
```

You can also terminate from a `while` loop using `break`:

```
a = 0

while true do
  p a
  a += 1

  break if a < 10
end

p a
```

This prints the numbers 0 and 1.

`break` accepts a value that supplies the result of the expression it is "breaking" out of:

```
result = [1, 2, 3].each do |value|
  break value * 2 if value.even?
end

p result # prints 4
```

`next` Statement¶ ↥

Use `next` to skip the rest of the current iteration:

```
result = [1, 2, 3].map do |value|
  next if value.even?

  value * 2
end

p result # prints [2, nil, 6]
```

`next` accepts an argument that can be used the result of the current block iteration:

```
result = [1, 2, 3].map do |value|
  next value if value.even?

  value * 2
end

p result # prints [2, 2, 6]
```

`redo` Statement¶ ↥

Use `redo` to redo the current iteration:

```
result = []

while result.length < 10 do
  result << result.length

  redo if result.last.even?

  result << result.length + 1
end

p result
```

This prints [0, 1, 3, 3, 5, 5, 7, 7, 9, 9, 11]

In Ruby 1.8 you could also use `retry` where you used `redo`. This is no longer true, now you will receive a SyntaxError when you use `retry` outside of a `rescue` block. See Exceptions for proper usage of `retry`.

Flip-Flop¶ ↥

The flip-flop is rarely seen conditional expression. It's primary use is for processing text from ruby one-line programs used with `ruby -n` or `ruby -p`.

The form of the flip-flop is an expression that indicates when the flip-flop turns on, `..` (or `...`), then an expression that indicates when the flip-flop will turn off. While the flip-flop is on it will continue to evaluate to `true`, and `false` when off.

Here is an example:

```
selected = []

0.upto 10 do |value|
  selected << value if value==2..value==8
end

p selected # prints [2, 3, 4, 5, 6, 7, 8]
```

In the above example the on condition is `n==2`. The flip-flop is initially off (false) for 0 and 1, but becomes on (true) for 2 and remains on through 8. After 8 it turns off and remains off for 9 and 10.

The flip-flop must be used inside a conditional such as `if`, `while`, `unless`, `until` etc. including the modifier

forms.

When you use an inclusive range (`..`) the off condition is evaluated when the on condition changes:

```
selected = []

0.upto 5 do |value|
  selected << value if value==2..value==2
end

p selected # prints [2]
```

Here both sides of the flip-flop are evaluated so the flip-flop turns on and off only when `value` equals 2. Since the flip-flop turned on in the iteration it returns true.

When you use an exclusive range (`...`) the off condition is evaluated on the following iteration:

```
selected = []

0.upto 5 do |value|
  selected << value if value==2...value==2
end

p selected # prints [2, 3, 4, 5]
```

Here the flip-flop turns on when `value` equals 2 but doesn't turn off on the same iteration. The off condition isn't evaluated until the following iteration and `value` will never be two again.