Aula 07

Operadores lógicos e precedência de operadores

Prof. Jalerson Lima

Apresentação

Nessa aula iremos dar continuidade aos estudos dos operadores em Ruby. Dessa vez vamos estudar sobre os operadores lógicos, que nos permitem realizar operações lógicas com valores booleanos. Além disso, como esta é a última aula sobre operadores, vamos aprender também sobre precedência de operadores.

Objetivos

1. Compreender os operadores lógicos;
2. Aprender como utilizar os operadores lógicos em Ruby;
3. Compreender como construir expressões lógicas;
4. Aprender o que é e como funciona a precedência de operadores.

1. Introdução

Vamos dar continuidade aos estudos dos operadores em Ruby, e dessa vez vamos abordar sobre os operadores lógicos. Os operadores lógicos nos permitem construir expressões complexas usando valores booleanos, e o retorno (resultado) é sempre um valor booleano. A linguagem Ruby possui sete desses operadores. Mas antes de começarmos a tratar sobre eles, vamos a alguns exemplos para nos recordar sobre a utilidade deles.

Um dos operadores lógicos é o `and`, que significa "e" em inglês. O operador lógico `and` retorna verdadeiro (*true*) apenas quando as duas proposições são verdadeiras (*true*), em todos os outros casos esse operador irá retornar falso (*false*). Para exemplificar o uso desse operador, considere o exemplo abaixo.

```
irb(main):001:0> a = true
=> true
irb(main):002:0> b = true
=> true
irb(main):003:0> c = false
=> false
irb(main):004:0> a and b
=> true
irb(main):005:0> a and c
=> false
irb(main):006:0> b and c
=> false
```

Observe que o operador `and` retornou verdadeiro (*true*) quando as duas variáveis eram verdadeiras (*true*). Quando pelo menos uma das variáveis era falsa (*false*), o retorno foi falso (*false*).

2. Operadores lógicos em Ruby

Ruby possui sete operadores lógicos que representam quatro operações lógicas.

Tabela 1 - Operadores lógicos em Ruby

Operador	Nome	Descrição
and	E	Retorna verdadeiro quando os dois operandos forem verdadeiros.
&&	E	Retorna verdadeiro quando os dois operandos forem verdadeiros.
or	Ou	Retorna verdadeiro quando pelo menos um dos dois operandos for verdadeiro.
||	Ou	Retorna verdadeiro quando pelo menos um dos dois operandos for verdadeiro.
^	Ou exclusivo	Retorna verdadeiro quando apenas um operando for verdadeiro. Quando os dois operandos forem verdadeiros, retorna falso.
!	Negador	Inverte o valor lógico do operando.
not	Negador	Inverte o valor lógico do operando.

A Tabela 1 apresenta todos os operadores lógicos disponíveis em Ruby. Observe que alguns deles representam a mesma operação lógica, como o `and` e o `&&`, o `||` e o `or`, o `!` e o `not`.

2.1 Operação lógica: and

A operação lógica *and* irá retornar verdadeiro apenas quando os dois operandos forem verdadeiros. Observe a tabela da verdade do operador `and` na Tabela 2.

Tabela 2 - Tabela da verdade do operador `and`

A	B	A and B
Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Falso
Falso	Verdadeiro	Falso
Falso	Falso	Falso

A operação lógica *and* pode ser executada utilizando um dos dois operadores lógicos: `and` ou `&&`. Observe alguns exemplos de uso desse operador no Exemplo de código 1.

```
1  puts "true and true = #{true and true}"
2  puts "true && false = #{true && false}"
3  puts "false and true = #{false and true}"
4  puts "false && false = #{false && false}"
5
6  gets
```

Exemplo de código 1 - Exemplos de uso do operador lógico `and`

2.2 Operação lógica: or

A operação lógica *or* irá retornar verdadeiro quando pelo menos um dos operandos for verdadeiro. Observe a tabela da verdade do operador `or` na Tabela 3.

Tabela 3 – Tabela da verdade do operador `or`

A	B	A or B
Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Verdadeiro
Falso	Verdadeiro	Verdadeiro
Falso	Falso	Falso

A operação lógica *or* pode ser executada utilizando um dos dois operadores lógicos: `or` ou `||`. Observe alguns exemplos de uso desse operador no Exemplo de código 2.

```
1  puts "true or true = #{true or true}"
2  puts "true || false = #{true || false}"
3  puts "false or true = #{false or true}"
4  puts "false || false = #{false || false}"
5
6  gets
```

Exemplo de código 2 – Exemplos de uso do operador lógico `or`

2.3 Operação lógica: ou exclusivo

A operação lógica 'ou exclusivo', também conhecida como *xor* (do inglês *exclusive or*), retorna verdadeiro quando apenas um dos operadores é verdadeiro, em todos os outros casos será retornado falso. Observe a tabela da verdade do 'ou exclusivo' na Tabela 4.

Tabela 4 – Tabela da verdade do 'ou exclusivo'

A	B	A xor B
Verdadeiro	Verdadeiro	Falso
Verdadeiro	Falso	Verdadeiro
Falso	Verdadeiro	Verdadeiro
Falso	Falso	Falso

A operação lógica *xor* pode ser executada utilizando o operador `^`. Observe os exemplos de uso desse operador no Exemplo de código 3.

```
1  puts "true ^ true = #{true ^ true}"
2  puts "true ^ false = #{true ^ false}"
3  puts "false ^ true = #{false ^ true}"
4  puts "false ^ false = #{false ^ false}"
5
6  gets
```

Exemplo de código 3 – Exemplos de uso do operador lógico `^`

2.4 Operação lógica: negador

A operação lógica de negação inverte o valor lógico do operando. Se o operando for verdadeiro, ele se tornará falso, e se for falso, se tornará verdadeiro. Observe a tabela da verdade do operador `not` na Tabela 5.

Tabela 5 – Tabela da verdade do operador `not`

A	not A
Verdadeiro	Falso
Falso	Verdadeiro

A operação lógica *not* pode ser executada utilizando um dos seguintes operadores: `not` ou `!`. Observe os exemplos de uso desse operador no Exemplo de código 4.

```
1  puts "not true = #{not true}"
2  puts "! false = #{! false}"
3
4  gets
```

Exemplo de código 4 – Exemplos de uso do operador lógico `not`

Atividade 7.1

Crie um *script* em Ruby com o código apresentado nos Exemplos de código 1, 2, 3 e 4. Execute o *script* e veja o resultado apresentado.

3. Expressões lógicas

Utilizando operadores lógicos e relacionais, nós somos capazes de construir expressões lógicas mais complexas. Confira os exemplos no Exemplo de código 5.

```
 1  a = true
 2  b = false
 3  c = true
 4  d = false
 5
 6  exemplo1 = (a and b) or (b and c)
 7  exemplo2 = (d or c) and not a
 8  exemplo3 = (a or b) or !c
 9  exemplo4 = (3 < 4) or (a ^ c)
10  exemplo5 = (10 >= 10) and (a or b) or not d
11
12  puts "(a and b) or (b and c) = #{exemplo1}"
13  puts "(d or c) and not a = #{exemplo2}"
14  puts "(a or b) or !c = #{exemplo3}"
15  puts "(3 < 4) or (a ^ c) = #{exemplo4}"
16  puts "(10 >= 10) and (a or b) or not d = #{exemplo5}"
17
18  gets
```

Exemplo de código 5 - Exemplos com operadores lógicos e relacionais

Atividade 7.2

Antes de executar o Exemplo de código 5, tente determinar o resultado de cada expressão lógica (linhas 6 a 10), depois execute o código e confira se você acertou.

Se você tentou determinar o resultado de cada expressão lógica da Atividade 7.2, você provavelmente deve ter achado algo estranho na expressão da linha 7, e provavelmente não sabe explicar porquê. A expressão é a seguinte: `(d or c) and not a`. Considerando que `d` é `false` e `c` é `true`, o resultado de `(d or c)` é `true`, e considerando que `a` é `true`, `not a` resulta em `false`. Por fim temos `true and false` que resulta em `false`. Contudo, na linha 13, o valor da variável `exemplo2`, que guarda o resultado da expressão da linha 7, é `true`! Como isso é possível?

Para complicar mais, se você teve a curiosidade de tentar executar a expressão da linha 7 no IRB, vai ver que o resultado da expressão é, de fato, *false*. Observe o exemplo abaixo.

```
irb(main):001:0> a = true
=> true
irb(main):002:0> b = false
=> false
irb(main):003:0> c = true
=> true
irb(main):004:0> d = false
=> false
irb(main):005:0> (d or c) and not a
=> false
irb(main):006:0> exemplo2 = (d or c) and not a
=> false
irb(main):007:0> exemplo2
=> true
irb(main):008:0>
```

No exemplo anterior, definimos os valores das variáveis `a`, `b`, `c` e `d`, conforme o Exemplo de código 5, e em seguida executamos a expressão `(d or c) and not a`, que resultou em `false`. Quando atribuímos o resultado dessa mesma expressão para a variável `exemplo2`, o valor dela é `true`. Como é possível que o resultado da expressão seja `false`, e o valor atribuído à variável seja `true`? A resposta para essa pergunta está na precedência de operadores.

4. Precedência de operadores

Para explicar a precedência de operadores, vamos usar um exemplo da matemática. Qual é o resultado da expressão aritmética abaixo?

$$3 \times 4 + 2$$

Se você respondeu que o resultado é 14, parabéns! Você acertou! E se você acertou, você levou em consideração a precedência de operadores. Mas o que é isso? A precedência de operadores determina quais operações serão executadas primeiro e quais serão executadas depois. Na matemática, sabemos que o operador de multiplicação tem precedência sobre o operador de soma, portanto devemos primeiro multiplicar e depois somar.

Assim como a matemática, as linguagens de programação também possuem precedência de operadores. A Tabela 6 apresenta operadores em ordem de precedência, ou seja, os operadores que aparecem no topo da tabela têm precedência (ou seja, executam primeiro) sobre os operadores que aparecem na parte de baixo na tabela.

Tabela 6 - Precedência de operadores em Ruby. Fonte: (W3RESOURCE, 2015)

Operador
!, ~ e +
**
-
*, / e %
+ e -
<< e >>
&
| e ^
>, >=, < e <=
<=>, ==, ===, !=, =~ e !~
&&
||
.. e ...
? e :
rescue
=, +=, -=, etc.
defined?
not
or e and
if, unless, while e until
{ } blocks

Vale salientar que, assim como na matemática, o uso de parênteses garante a precedência de execução da expressão que está entre os parênteses. Um exemplo clássico é a fórmula para calcular a média aritmética de dois números (ilustrada abaixo).

$$media = (a + b)/2$$

Os parênteses garantem que a soma $a + b$ será executada antes da divisão por 2, e isso é necessário porque a divisão tem precedência sobre a soma.

Para explicar porque a variável `exemplo2` ficou com o valor `true` quando recebeu o resultado da expressão `(d or c) and not a` no Exemplo de código 5, precisamos levar em consideração a precedência de operadores. Observe, na Tabela 6, que o operador de atribuição (`=`) tem precedência sobre os operadores `not`, `or` e `and`, portanto a atribuição é executada primeiro.

Agora vamos repensar a execução de `exemplo2 = (d or c) and not a` levando em consideração a precedência de operadores. A primeira parte da expressão a ser executada é a `(d or c)` porque ela está entre parênteses, e o resultado será `true`. Pela precedência de operadores, o próximo operador a ser executado é a atribuição, e é por isso que a variável `exemplo2` recebe o valor `true` (resultado de `d or c`). O próximo operador a ser executado é o `not`, portanto `not a` resulta em `false`. Por fim, o que resta da expressão é `true and false`, que resulta em `false`.

Isso explica porque o resultado da expressão `(d or c) and not a` é `false`, mas quando se introduz o sinal de atribuição (`exemplo2 = (d or c) and not a`), a variável `exemplo2` irá receber `true`, que é o resultado de apenas parte da expressão (`d or c`), e não dela toda.

Resumindo

Essa aula finaliza a apresentação dos operadores. Nela compreendemos como funcionam e como utilizar os operadores lógicos, inclusive para construção de expressões que envolvam também operadores relacionais. Além disso, aprendemos o que é e como funciona a precedência de operadores em Ruby.

Referências

POINT, T. Ruby Operators. **Tutorials Point**, 2015. Disponivel em: <http://www.tutorialspoint.com/ruby/ruby_operators.htm>. Acesso em: 03 nov. 2015.

SOUZA, L. **Ruby - Aprenda a programar na linguagem mais divertida**. 1ª. ed. São Paulo: Casa do Código, v. I, 2012.

W3RESOURCE. Ruby Operators Precedence. **W3Resource**, 2015. Disponivel em: <http://www.w3resource.com/ruby/ruby-operators-precedence.php>. Acesso em: 23 nov. 2015.