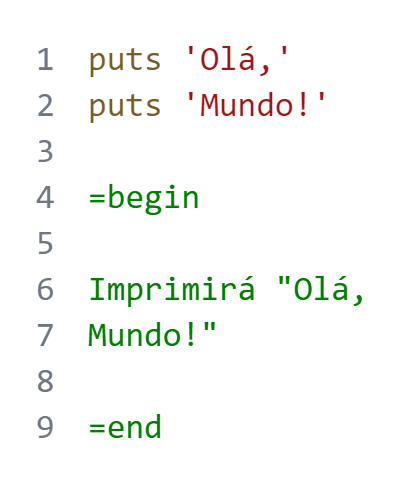
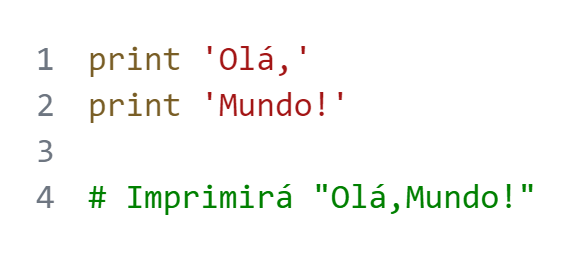
# ‘puts’ e ‘print’

Ambos são usados para imprimir uma mensagem. O comando **‘puts’** imprime a mensagem começando de uma nova linha.

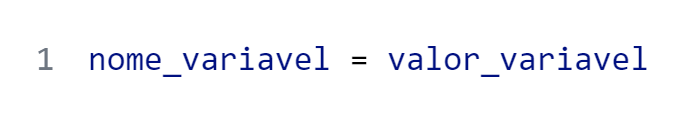


Já o **‘print’** imprime sem a quebra de linha, continuando a partir do último caractere.



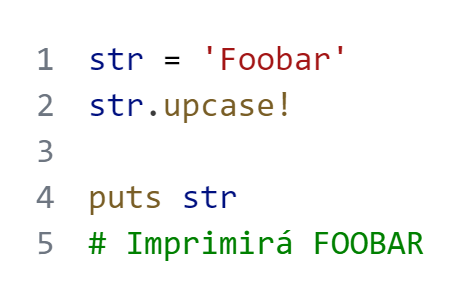
# Variáveis

Em Ruby, sua sintaxe é a seguinte:

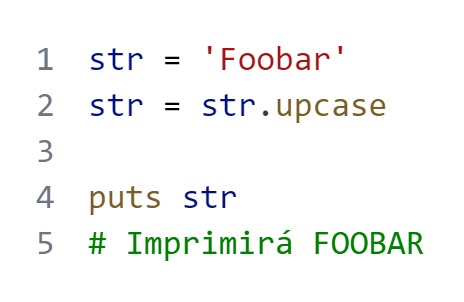
São comumente escritas em minúsculas e em caso de nomes compostos, é utilizado o underline (**‘\_’**).

## Substituir com Métodos

Quando necessário adicionar um método a uma variável, substituindo a variável anterior, podemos adicionar uma exclamação (**‘!’**) ao final da declaração.

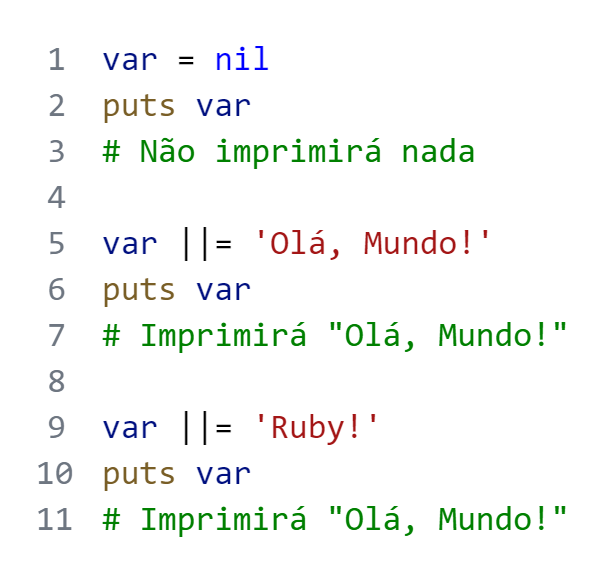


Seria o mesmo que:



## Atribuição Condicional

O operador **‘||=’** é a junção dos operadores **‘ou’** (**‘||’**) e o operador de **atribuição** (**‘=’**) são usados para atribuir o algum valor a uma variável apenas se essa variável ainda não possui algum valor – ou seja, se a mesma for igual a **‘nil’**.

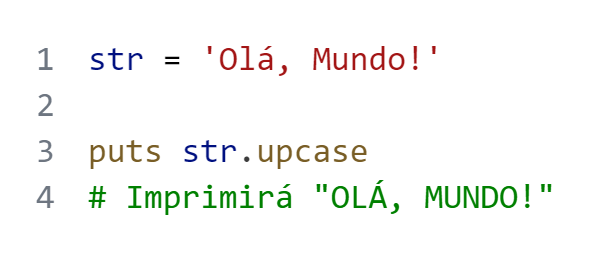
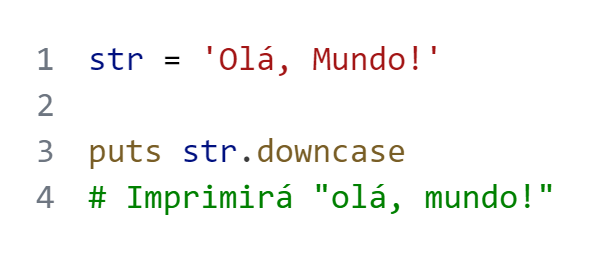
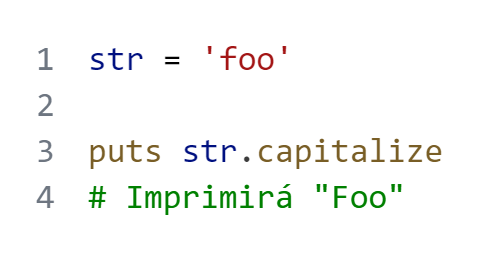
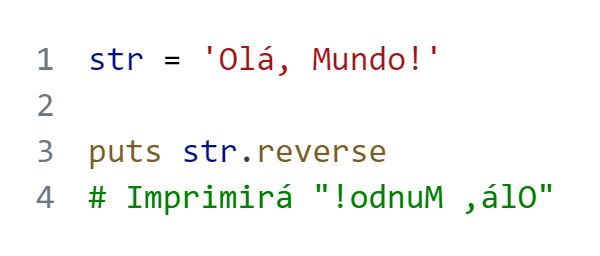
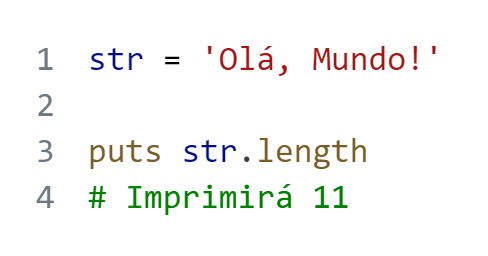
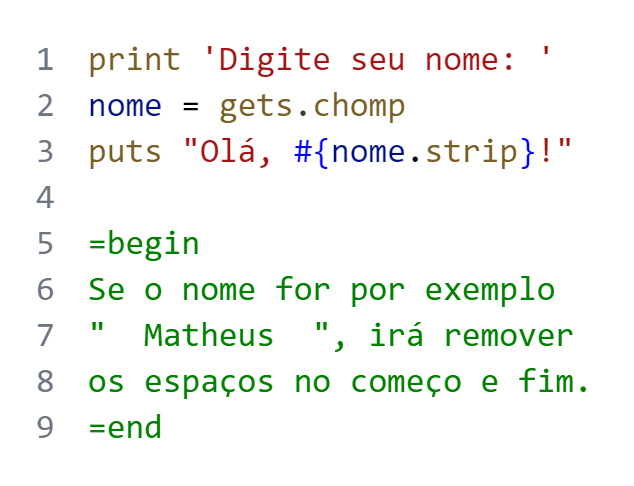


# Tipos de Dados

## String

Tudo entre aspas simples ou duplas são considerados strings.

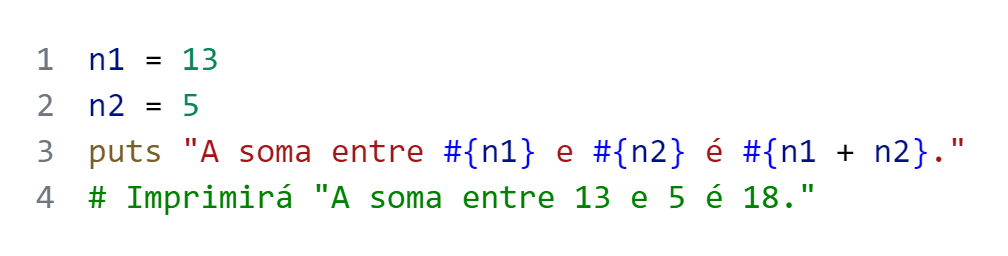
### Métodos de String

* **‘.upcase’:** Transforma todas as letras da string em maiúsculas.  
  
* **‘.downcase’:** Transforma todas as letras da string em minúsculas.  
  
* **‘.capitalize’:** Transforma primeira letra da string em maiúscula e o resto em minúsculas.  
  
* **‘.reverse’:** Inverte o texto da string.  
  
* **‘.length’:** Retorna a quantidade de caracteres que a string possui.  
  
* **‘.strip’:** Remove os espaços em branco no começo e no fim da string.  
  

### Interpolação

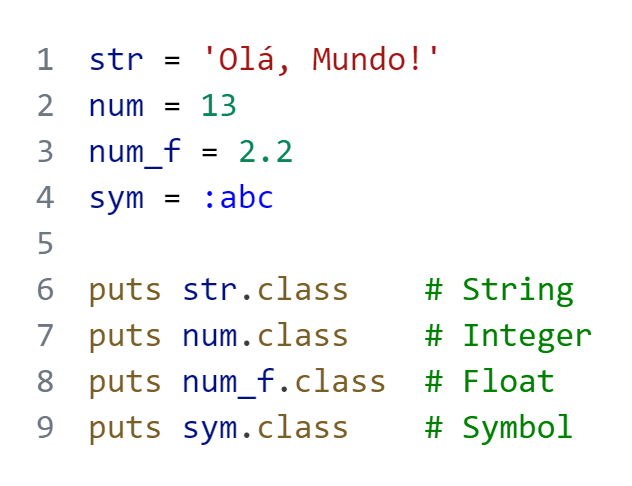
A interpolação de string permite adicionar variáveis ou expressões dentro da string.

Em Ruby, é feita por meio de chaves após o caractere hash (**‘#’**). A interpolação é feita somente com o uso entre aspas duplas.



## Conversão de Tipos

Primeiramente, devemos saber como verificar o tipo de um objeto específico. Para isso, existe o método **‘.class’**.

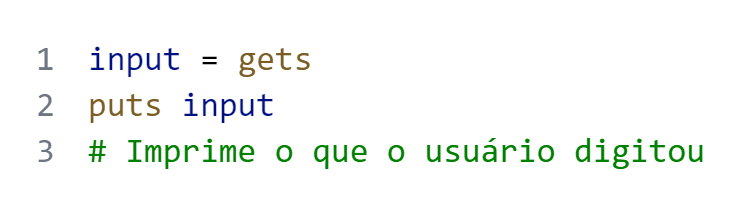


### Conversões

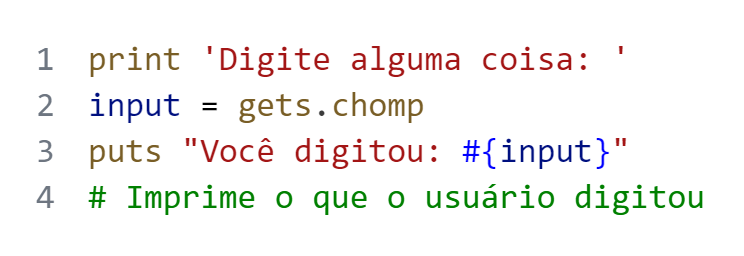
* **‘.to\_i’ / ‘.to\_int’:** Converte para Inteiro
* **‘.to\_f’:** Converte para Float.
* **‘.to\_s’:** Converte para String.
* **‘.to\_sym’ / ‘.intern’:** Converte para Símbolo.
* **‘.to\_c’:** Converte para Complexo.
* **‘.to\_enum’:** Converte para Enumerador.
* **‘.to\_a’:** Converte para um array.

# Input

Para que o usuário possa interagir por meio de uma entrada, existe o comando **‘gets’**.



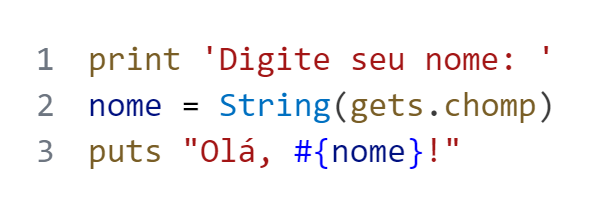
O comando ‘puts’ cria o campo de entrada em uma nova linha. Para que o campo seja sem criar uma nova linha, use **‘gets.chomp’**.



**Lembre-se**: Deve ser utilizado o **‘print’** ao invés do ‘puts’ para que não haja uma quebra de linha por parte do comando de impressão.

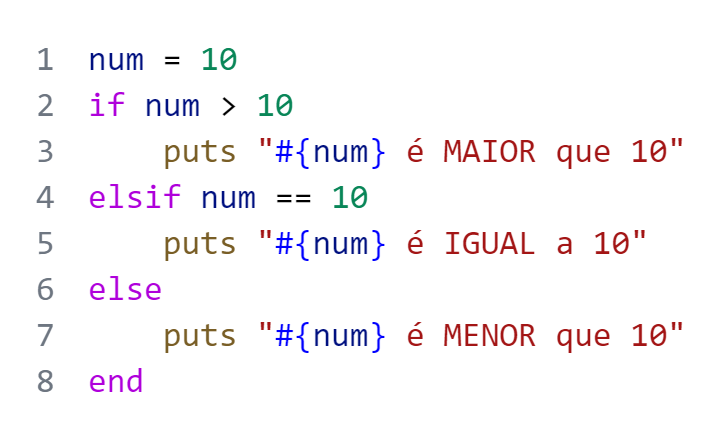
## Input pré-formatado

Caso o valor a ser passado pelo usuário deva ser uma string, número inteiro ou número flutuante, podemos definir ao início da declaração antes do ‘gets’.



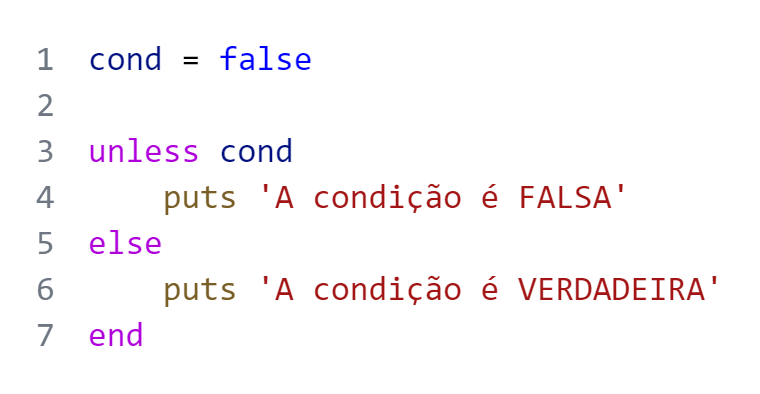
# Condições

Em Ruby, são: **‘if’/‘unless’**, **‘else’** e **‘elsif’**.



## ‘unless’

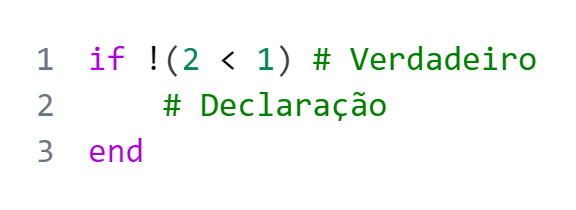
O **‘unless’** é o oposto do ‘if’, ou seja, ele executa o bloco abaixo caso sua condição seja falsa.



**OBS.**: Não pode haver nenhum **‘elsif’** em estruturas ‘unless’.

## Operadores Booleanos

Os operadores booleanos são:

* ‘&&’: **and**. Todas as expressões devem ser verdadeiras
* **‘||’:** **or**. Pelo menos uma das expressões deve ser verdadeira.
* **‘!’:** **not**. Se a expressão for verdadeira, ela se torna false, e vice-versa.  
  

## Comparadores

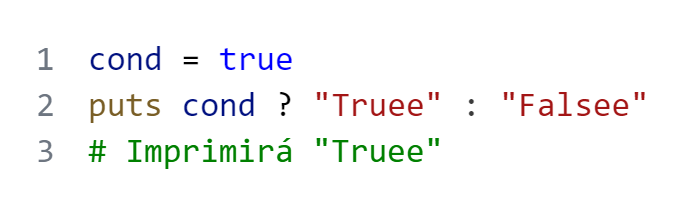
* **‘>’:** Maior.
* **‘<’:** Menor.
* **‘==’:** Igual.
* **‘>=’:** Maior ou igual.
* **‘<=’:** Menor ou igual.
* **‘!=’:** Diferente.

### Operador de Comparação Combinada

O operador de combinação combinada, representado por **‘<=>’**, retorna **1** caso o valor da esquerda seja maior que o valor da direta e retorna **-1** em caso contrário. Em caso de valores iguais, retorna **0**.

## Condicional Ternária

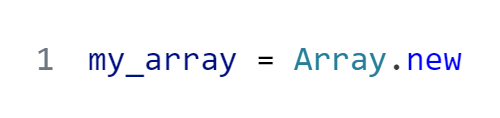
A Expressão Condicional Ternária é uma forma de usar ‘if’ e ‘else’ de forma mais simples e compacta. A declaração pode é feita em apenas uma linha.



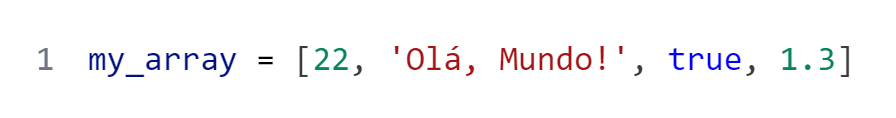
Primeiro o valor **booleano**, depois considere **‘?’** para **‘if’** e **‘:’** para **‘else’**.

# Arrays

Podem ser declaradas de duas formas:

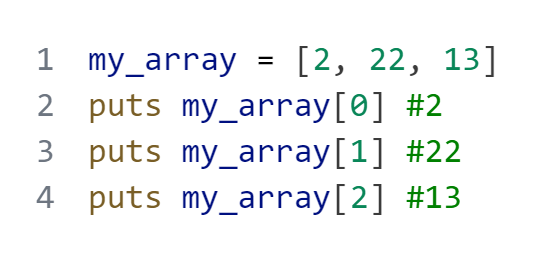
1. **Declaração literal:**
2. **‘Array.new’:**

Exemplo de Array:



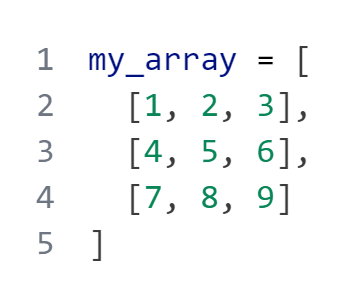
## Acesso aos elementos do Array

É usada a Notação de Colchetes junto aos índices de cada valor para acessar cada elemento de um array.



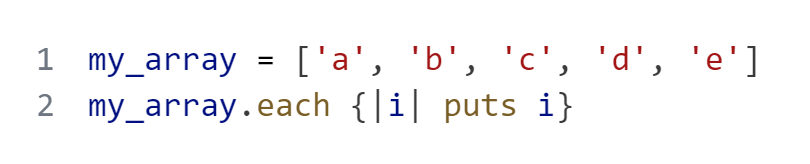
## Array Multidimensional

Assim se chama um array que tem como valores outros arrays. Por exemplo:



## Iterando sobre a Array

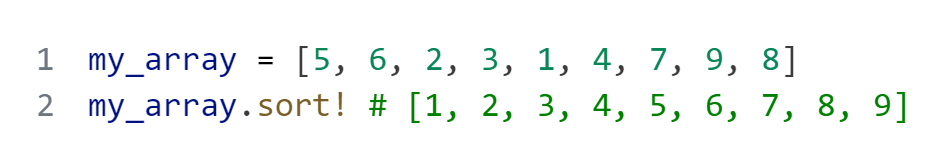
Com o método **‘each’**, é possível iterar sobre uma array. Deve ser passado um nome para referência de cada item dentro do array.



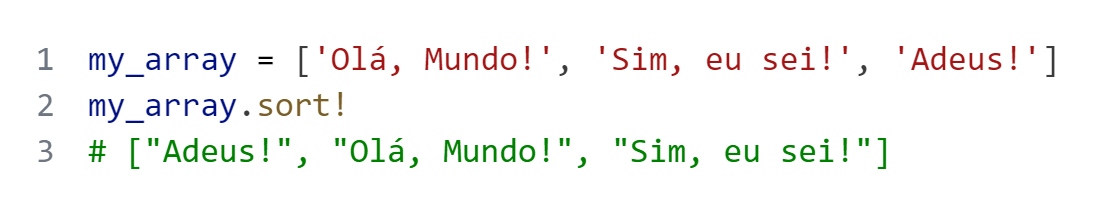
## Ordenando uma Array

Para ordenar um array em ordem alfabética ou numérica, é usado o método **‘sort’**.

No caso de números:

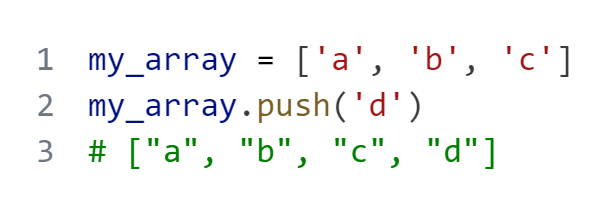
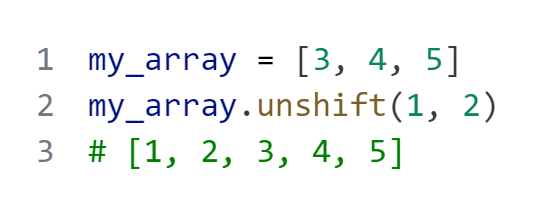
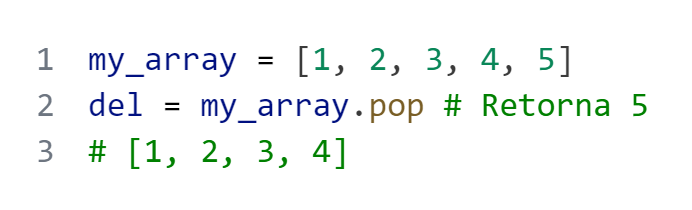
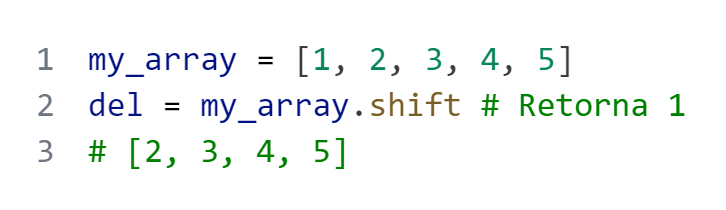
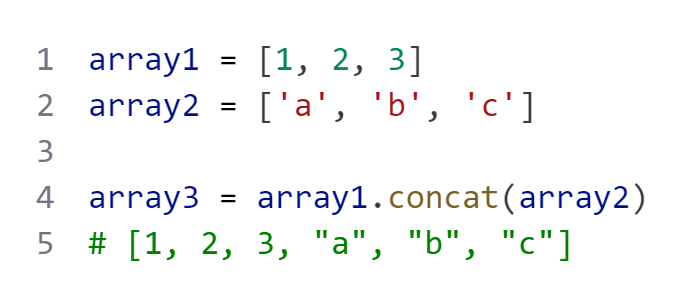
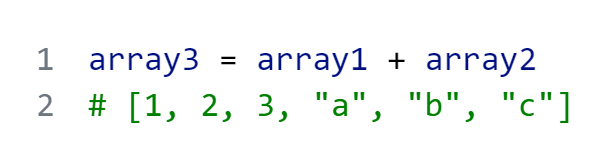
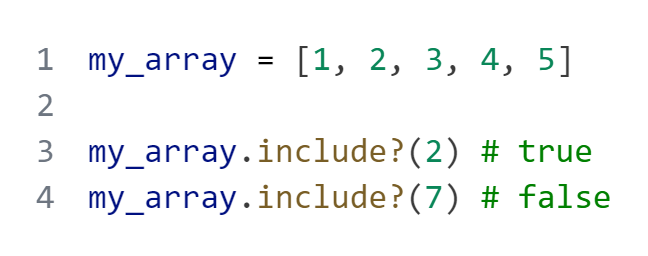
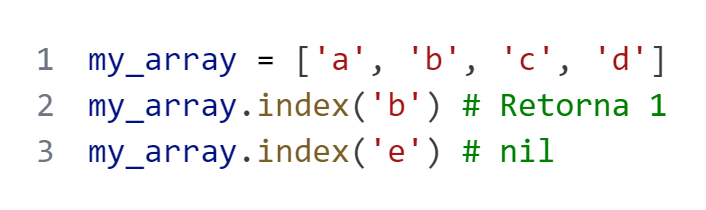
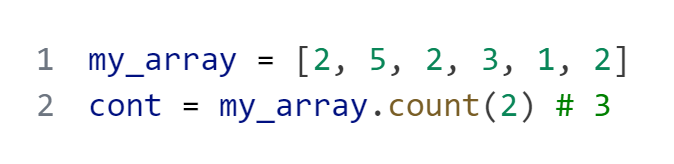
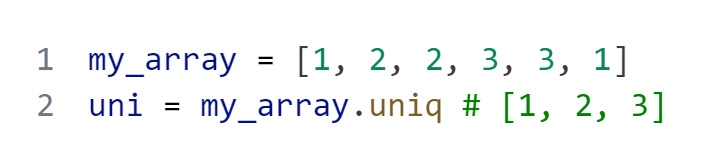
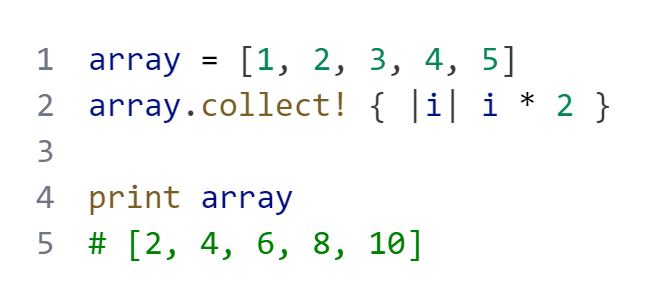
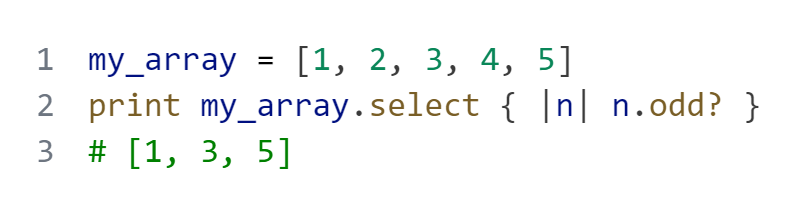
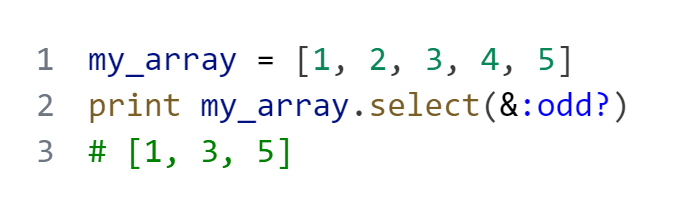


No caso de strings:



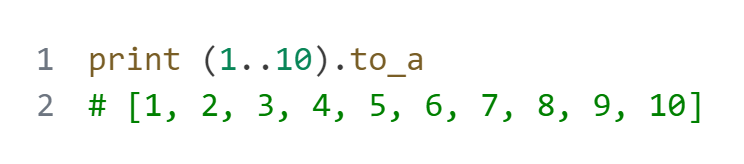
**OBS.**: Caso em um mesmo array exista números e strings, ocorrerá um erro (**‘ArgumentError’**) ao aplicar o método **‘sort’**.

## Métodos para Arrays

* **‘.push’:** Adiciona um ou mais elementos ao final do array.  
  
* **‘.unshift’:** Adiciona um ou mais elementos ao início do array.  
  
* **‘.pop’:** Exclui o último elemento do array e o retorna.  
  
* **‘.shift’:** Exclui o primeiro elemento do array e o retorna.  
  
* **‘.concat’:** Concatena um ou mais arrays a um array específico. Não altera o array original.  
    
  Seria o mesmo que usar o operador **‘+’**.  
  
* **‘.include?’:** Verifica se um elemento está presente dentro do array. Se presente, retorna verdadeiro, em caso contrário retorna falso.  
  
* **‘.index’:** Retorna o índice a qual se encontra o elemento especificado. Caso o elemento não esteja presente dentro do array, retorna **‘nil’**.  
  
* **‘.count’:** Retorna a quantidade de aparições de elementos com o mesmo valor.  
  
* **‘.uniq’:** Retorna um array com elementos únicos.  
  
* **‘.collect’ ou ‘.map’:** Aplica a expressão de um bloco a cada elemento de um array.  
  
* **‘.select’:** Cria um novo array apenas com os elementos que testam verdadeiro para uma condição específica.  
    
  Seria o mesmo que:  
  

## Conversão para Array

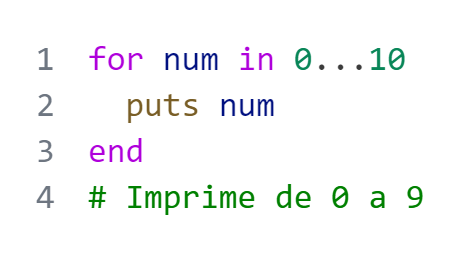
O método **‘to\_a’** converte um objeto para um array.



# Repetições e Iterações

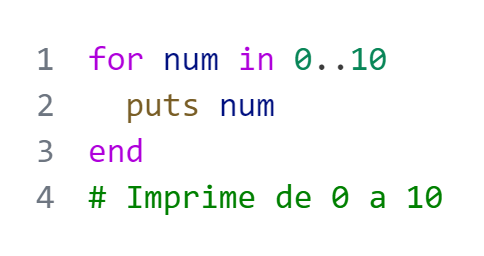
## ‘for’

Repete um block uma quantidade específica de vezes.



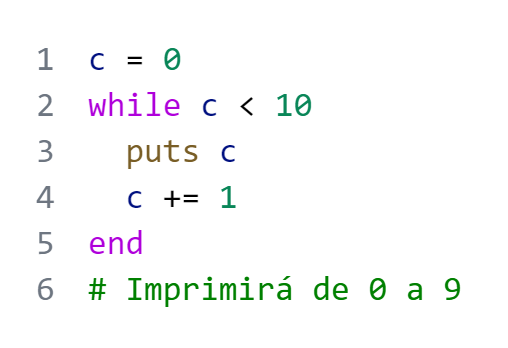
Caso use **‘...’**, ele fará a repetição até um número antes do último, como no caso acima, em que repetiu até o 9 ao invés de 10.

Para repetir até o maior número especificado, use **‘..’**.



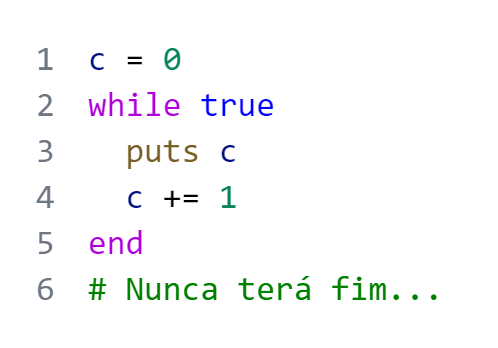
## ‘while’

Repete um bloco enquanto sua condição for verdadeira.



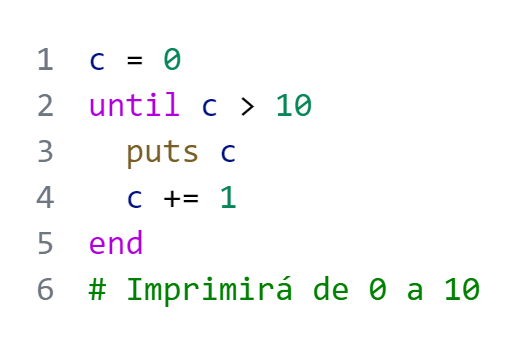
### ‘while’ infinito

Para um **‘while’** infinito, podemos usar uma condição que nunca será falsa, como ‘1 < 2’, ou simplesmente **‘true’**.



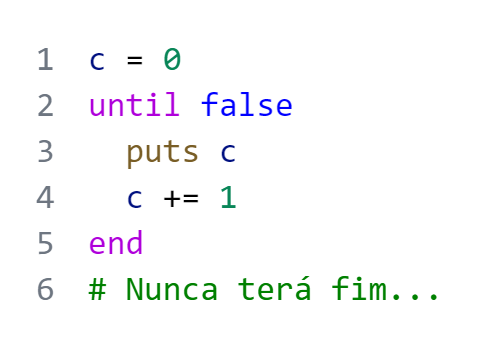
### ‘until’

É como o **‘while’**, porém ao contrário, pois repete um bloco até que uma condição se torne falsa.



#### ‘until’ infinito

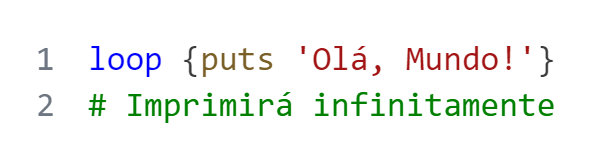
Ao contrário de um **‘while’** infinito, aonde se usa ‘true’ como condição, usa-se **‘false’** para um **‘until’** **infinito**.



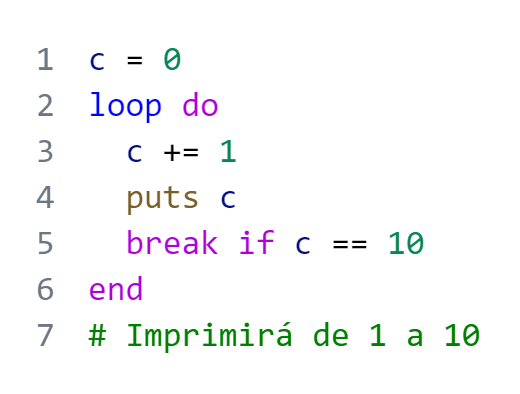
## ‘do’ e ‘{}’

Ambos são usados para abrir um bloco de código, mas o **‘do’** é mais comum quando o bloco possui mais de uma linha e é necessário **‘end’** para fechar seu bloco.

### ‘{}’ loop



### ‘do’ loop

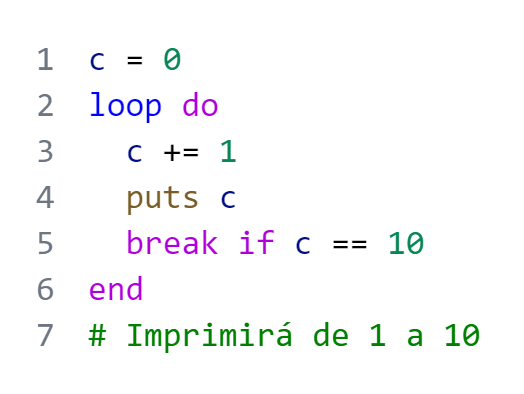


### Mais sobre ‘do’

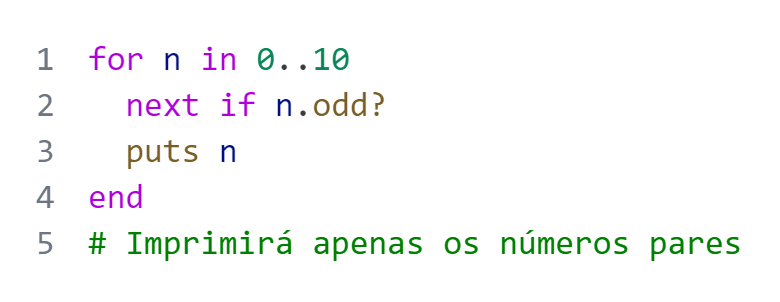
O **‘do’** pode ser usado com repetições, por exemplo **‘while’**, **‘until’** e **‘for’**, e métodos de iteração, como **‘each’** e **‘map’**.

## ‘break’ e ‘next’

A declaração **‘break’** interrompe imediatamente uma repetição.

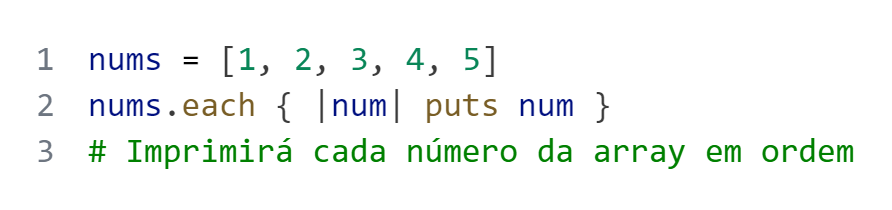


Já a declaração **‘next’** pula para a próxima repetição.

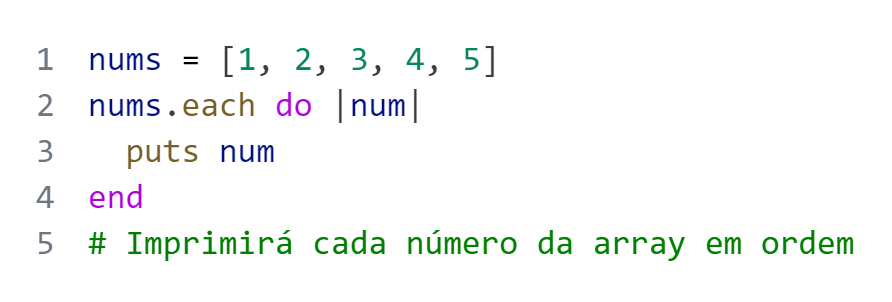


## Método ‘each’

Repete um bloco em uma quantidade de vezes relativa ao objeto em que foi aplicado. Se aplicado a uma string, repetirá na mesma quantidade de caracteres. Se aplicado a um array, repetirá na mesma quantidade de itens.

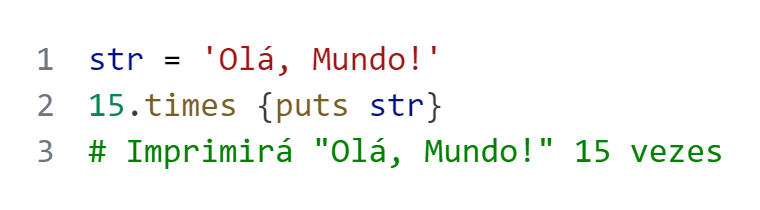


O mesmo seria:

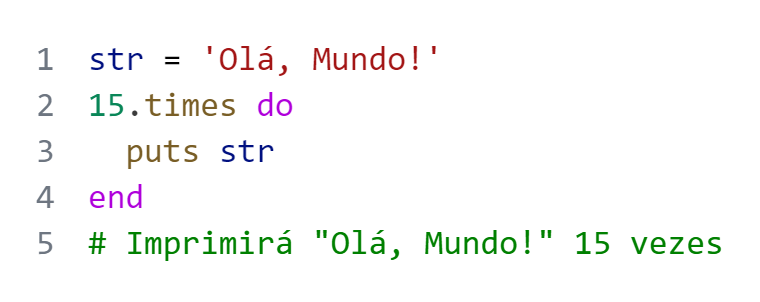


## Iterador ‘times’

Executa um bloco em uma quantidade específica de vezes. É como um **‘for’**, porém mais simples e compacto.

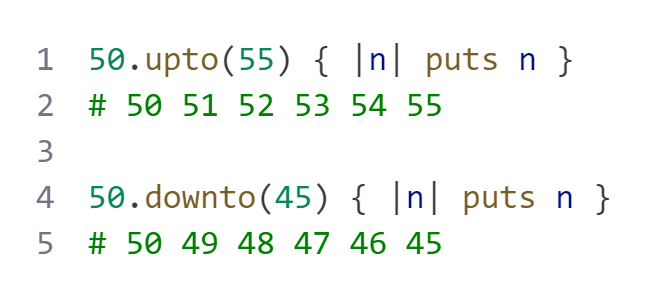


O mesmo seria:

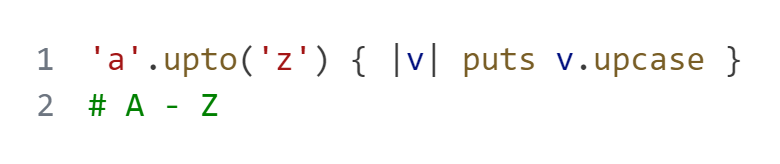


## Iteradores ‘upto’ e ‘downto’

Permitem iterar sobre valores específicos em uma sequência, sendo **‘upto’** para iterar de forma crescente e **‘downto’** para decrescente.

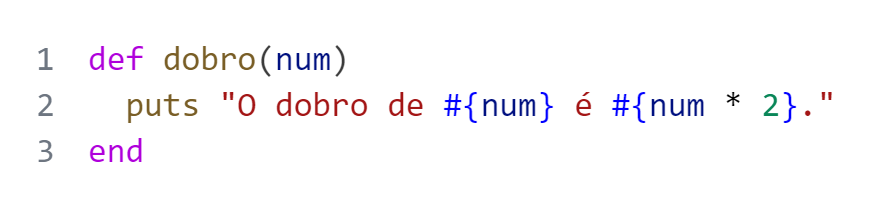


Dessa forma, é possível iterar até mesmo sobre letras específicas.



# Funções

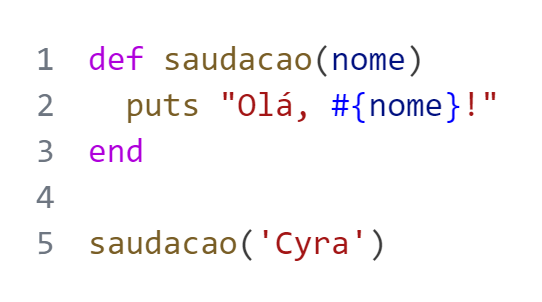
Em Ruby, as funções são criadas a partir da palavra-chave **‘def’**, e após dela deve vir o nome e parâmetros (opcional) da função.



É necessário que haja **‘end’** ao final da função para sinalizar o fim.

## Chamando Funções

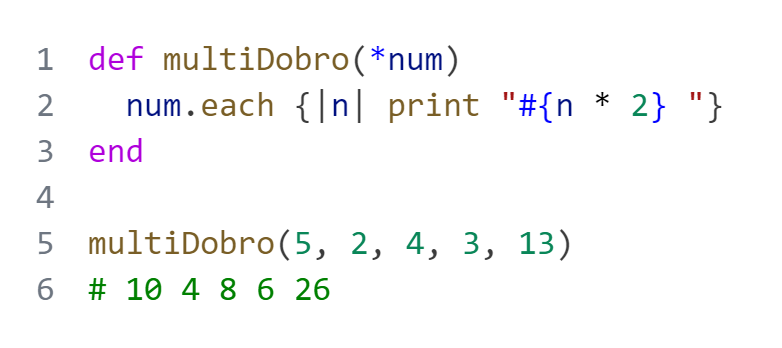
Simplesmente escrevendo o nome da função, ela será chamada.



Não há a necessidade do uso de parênteses caso a função não possua parâmetros.

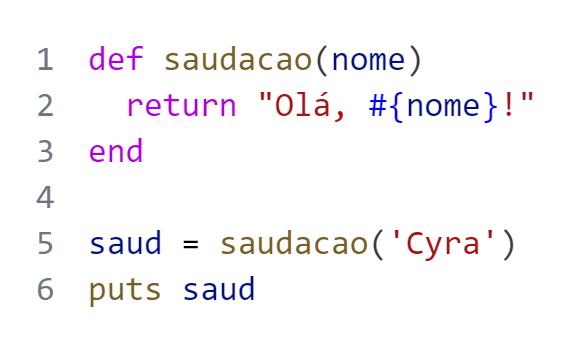
## Múltiplos Argumentos

Caso o número de argumentos seja indefinido, podendo não ser a mesma quantidade em diferentes execuções da função, é possível fazer com que esses argumentos vão direto para um array. Para isso, apenas coloque um asterisco (**‘\*’**) antes do nome do parâmetro.



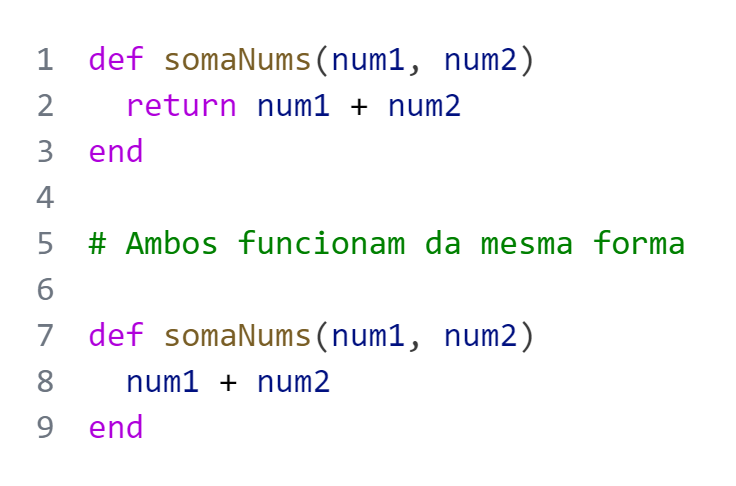
## Retorno

A declaração **‘return’** retorna algum valor, encerrando a função no mesmo momento.



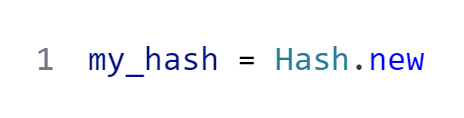
### Retorno Implícito

Em Ruby, é retornado automaticamente a última expressão calculada dentro da função.

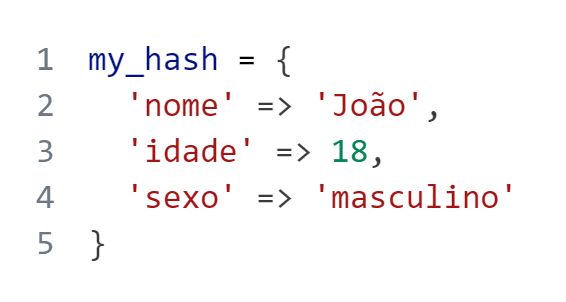


# Hashes

Há duas formas de declarar um hash:

1. **Declaração literal:**  
   
2. **‘Hash.new’:**

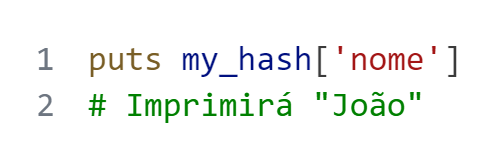
Exemplo de Hash:



‘nome’, ‘idade’ e ‘sexo’ são **chaves** (**keys**). ‘João’, 18 e ‘masculino’ são os **valores** (**values**).

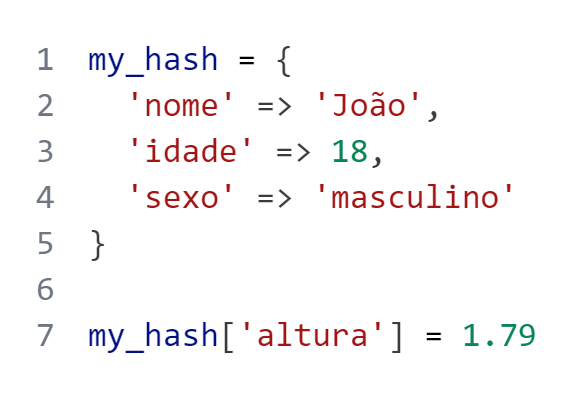
## Acesso aos dados do Hash

Para acessar os dados de um hash, pode-se usar Notação de Colchetes.

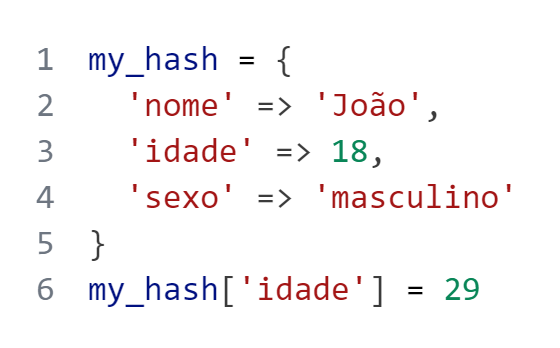


## Adição/Substituição de Dados

Para **adicionar** **dados** a um hash, você pode apenas declarar isso usando o nome do hash junto ao nome da chave por meio da notação de colchetes. Em seguida atribua o valor.

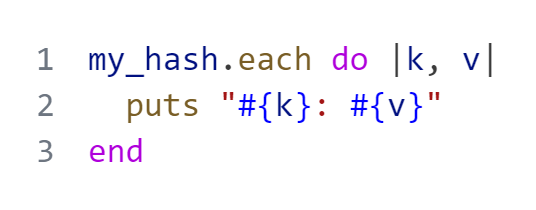


O mesmo vale caso você queira **substituir** o valor de uma chave já existente.



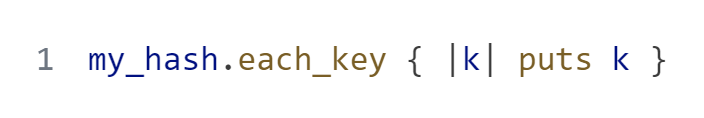
## Iterando sobre o Hash

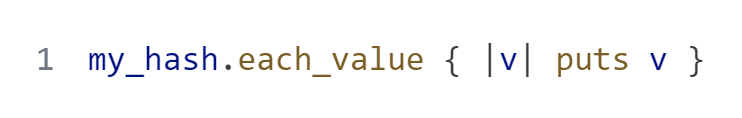
Utilizando o método **‘each’** e passando os nomes para as variáveis relativas às chaves e aos valores, podemos iterar sobre o hash.



### Iterar apenas sobre chave/valor

Os métodos **‘each\_key’** e **‘each\_value’** iteram apenas sobre as chaves e valores.

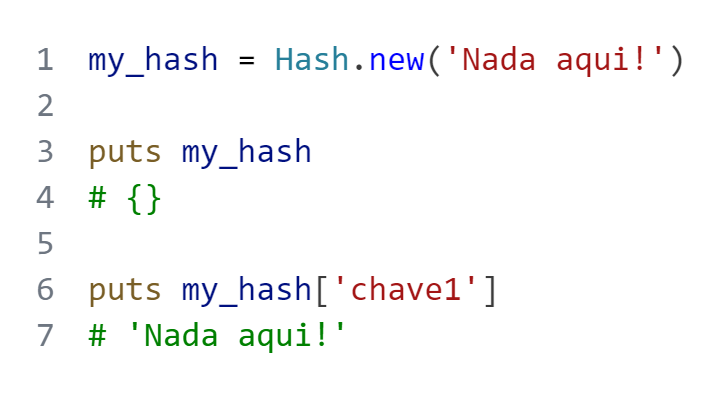




## Valor padrão para chaves

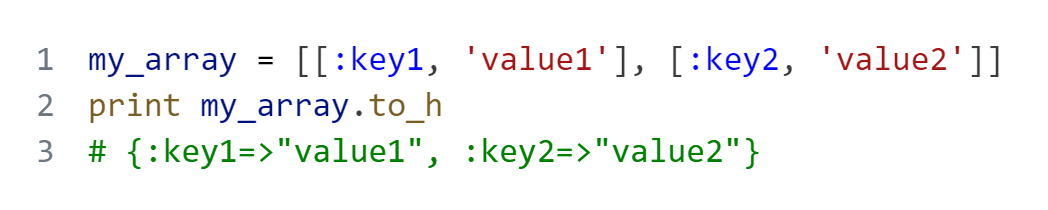
Caso o haja a solicitação de uma chave inexistente, Ruby irá retornar **‘nil’**, que significa **‘absolutamente nada’**.

Para definir um valor padrão para futuras chaves, adicione use **‘Hash.new(<valor\_padrao>)’**.



## Converter para Hash

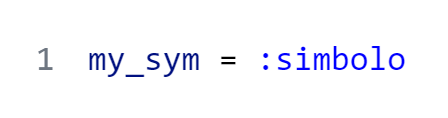
O método **‘to\_h’** permite a conversão de certos objetos para hash.



# Símbolos

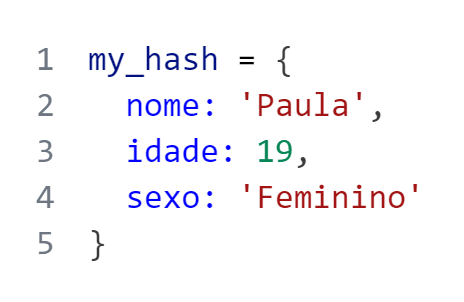
São nomes dados para referenciar alguma coisa, assim como chaves string em hashes.

Não devem estar envoltos em aspas, mas no começo deve haver dois pontos (**‘:’**).



## Símbolos como Chave Hash

Os símbolos podem ser usados como chaves em um Hash. Nesse caso, simplifica ainda mais a sintaxe do hash.

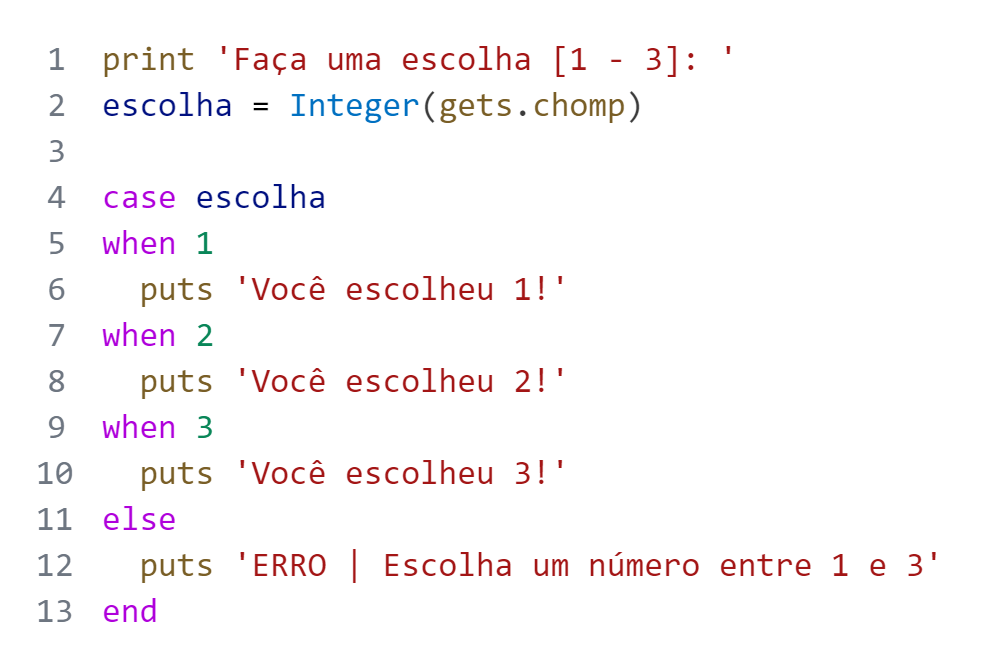


Perceba que não existe mais **‘=>’** para atribuir o valor à chave, pois o mesmo foi substituído por dois pontos (**‘:’**).

**DESEMPENHO**: Símbolos são mais rápidos que strings.

# Declaração ‘case’

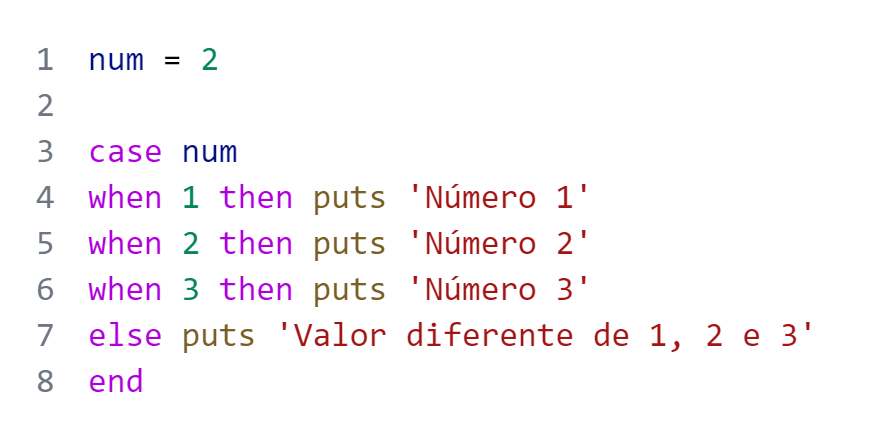
A declaração **‘case’** decide qual bloco executar de acordo com o valor de uma variável. Dentro da estrutura ‘case’, podem ser escritos um ou mais **‘when’**.



O bloco de **‘else’** será executado caso nenhum ‘when’ acima possua o valor igual à variável ‘escolha’.

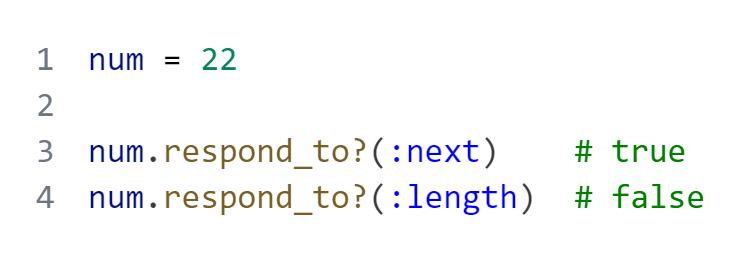
## ‘then’

Permite escrever uma declaração **‘when’** em apenas uma linha.



# Método ‘respond.to?’

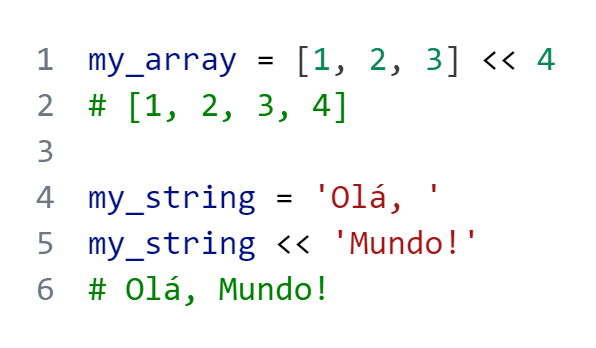
Verifica se algum método pode ser usado com o objeto em questão, retornando verdadeiro caso possa ser usado, ou falso em caso contrário.



O método deve ser escrito em forma de símbolo, ou seja, **‘:next’** ao invés de ‘.next’.

# “A pá” (The Shovel)

O operador **‘<<’** é usado para concatenação.

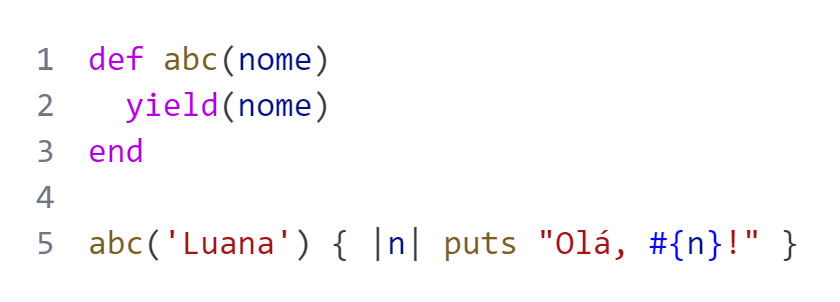


No caso de arrays, funciona como o método **‘push’**.

# Métodos *Yield*

Assim são chamados os métodos que aceitam um bloco. Esses métodos usam esses blocos para transferir o controle de chamada para o bloco e vice-versa.

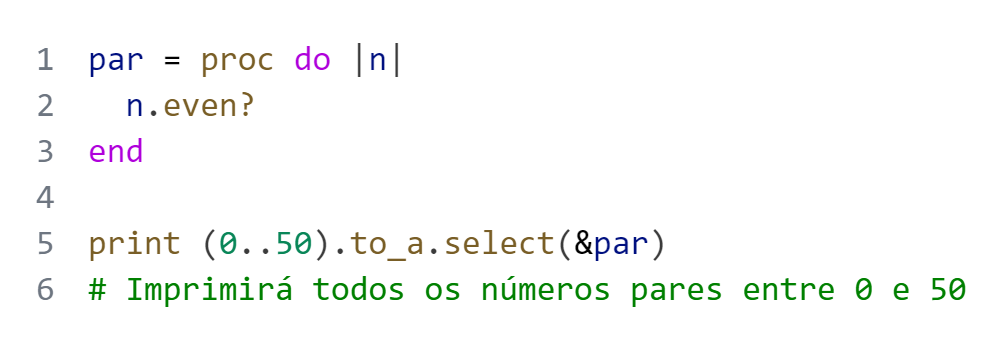
Exemplos são ‘.each’ e ‘.collect’, mas também é possível criar métodos desse tipo.



# Procs e Lambdas

## Procs

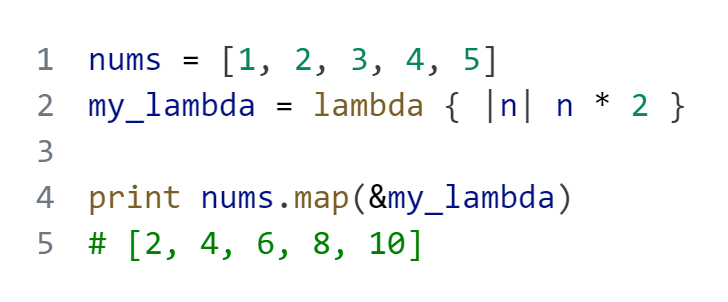
São blocos reutilizáveis salvos em uma variável.



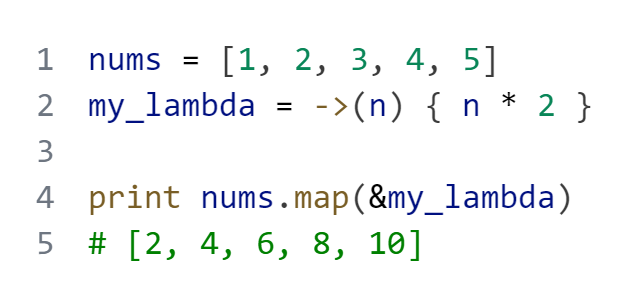
O **‘&’** converte o proc em um bloco, então deve ser usado sempre que um método espera um bloco como argumento.

## Lambdas

Assim como Proc, uma Lambda é um bloco armazenado dentro de uma variável, e sua sintaxe é inclusive a mesma, trocando apenas ‘proc’ por **‘lambda’** (ou **‘->’**).



Podemos substituir ‘lambda’ por **‘->’** e passar o elemento entre parênteses. O resultado será o mesmo.

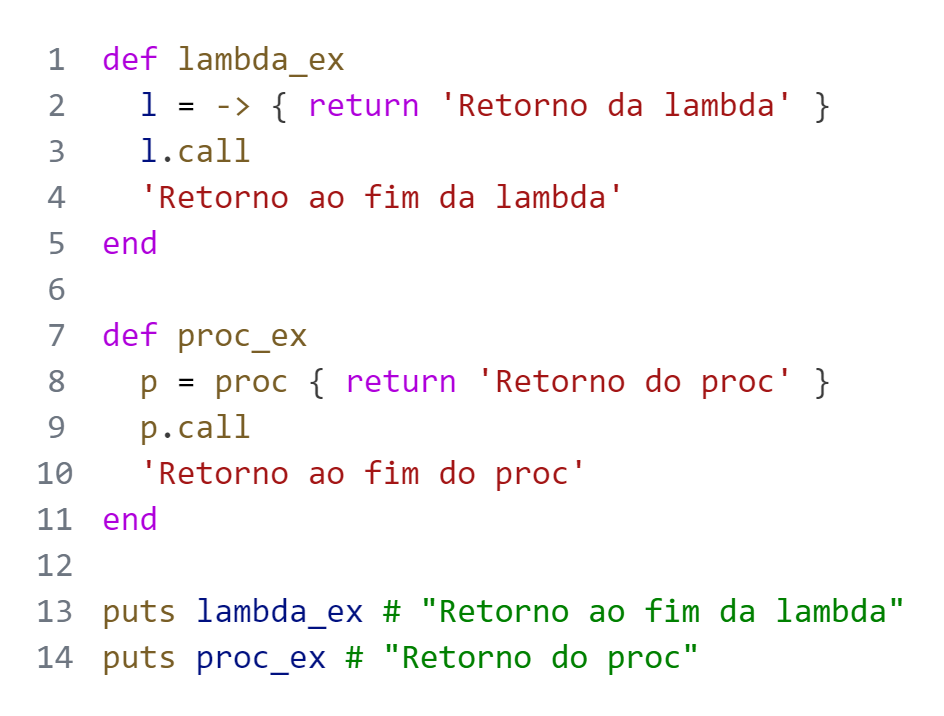


## Procs vs. Lambdas

Procs e Lambdas possuem duas principais diferenças:

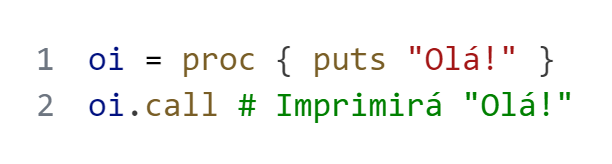
Primeiro, lambdas verificam o número de argumentos passados a ela, ou seja, um lambda lançará um erro caso passado o número errado de argumentos, enquanto um proc ignorará argumentos inesperados e atribuirá ‘nil’ a qualquer um que estiver faltando.

E segundo, ao retornar (‘return’) as lambdas não interrompem o método, ao contrário dos procs, que interromper o método assim que retornam um valor.



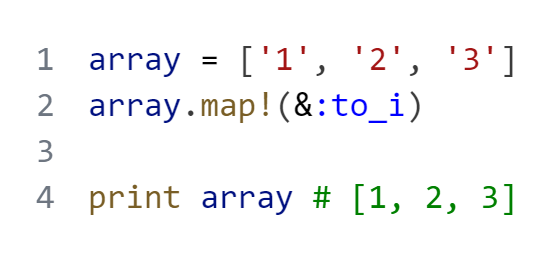
## Método ‘.call’

O método **‘call’** permite chamar procs e lambdas de forma simples.



## Métodos para Blocos

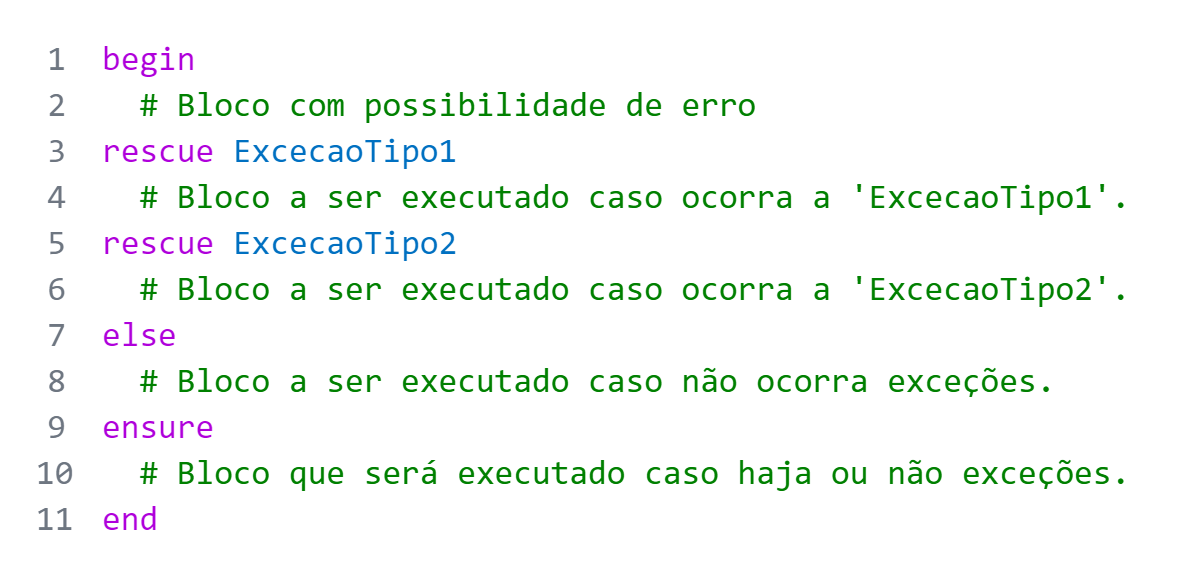
Usando o nome de um método e adicionando **‘:’** ao início, fará com que se torne um símbolo. Adicionar **‘&’** antes desse símbolo fará com que se torne um bloco. Então fica: **‘&:<nome\_metodo>’**.



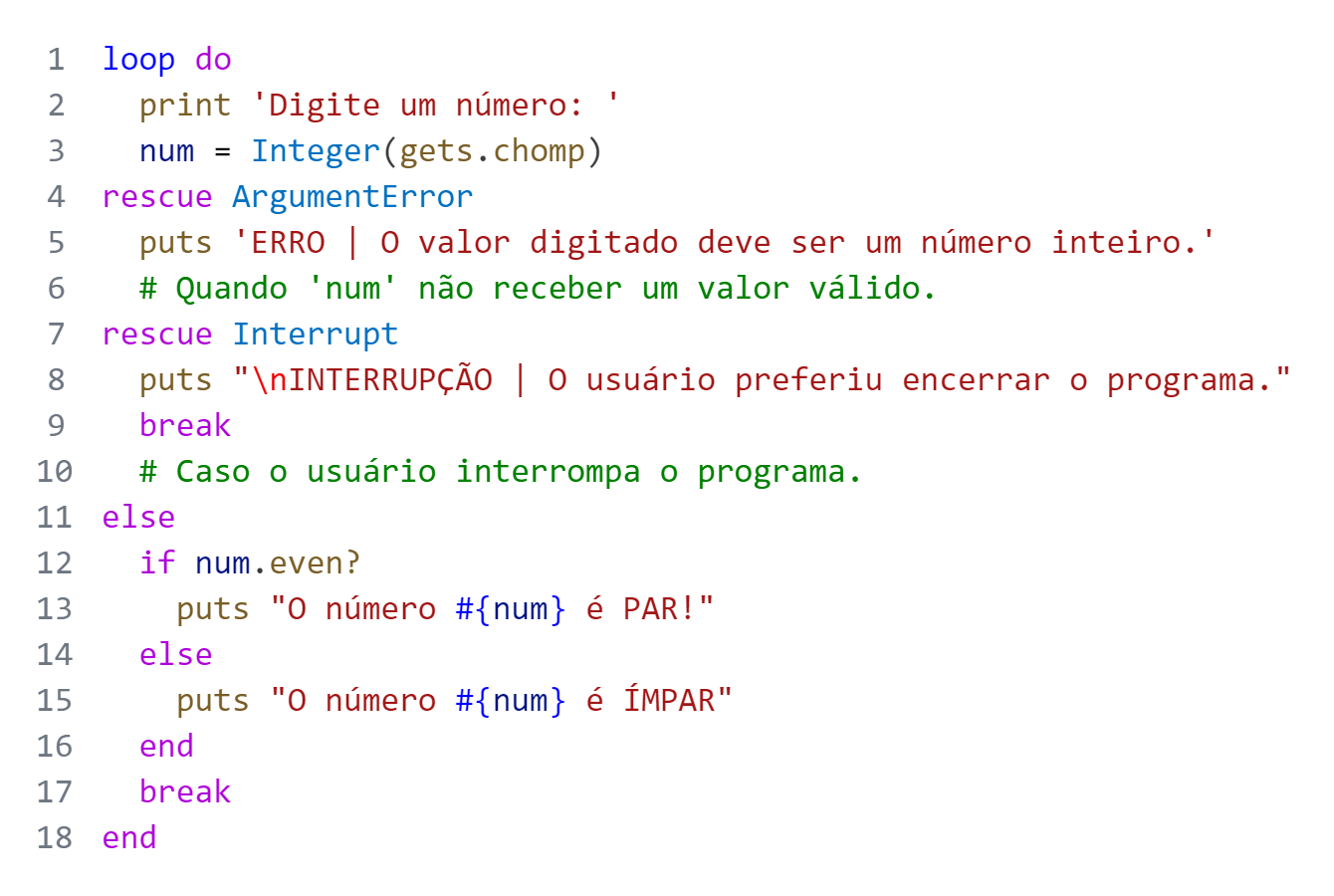
# Tratamento de Erros

Para tratar os erros são usados os blocos **‘begin’**, **‘rescue’**, **‘else’** e **‘ensure’**.

* **‘begin’:** Contém o trecho de código que pode ocorrer alguma exceção.   
  É apenas necessário caso o bloco em que pode ocorrer a exceção já não esteja dentro de um bloco.
* **‘rescue’:** Executa um bloco ao capturar uma exceção.
* **‘else’:** Este bloco executa caso não ocorra nenhuma exceção.
* **‘ensure’:** Este bloco executa sempre, mesmo que haja ou não exceções.



Abaixo um programa de exemplo que verifica se um valor é par ou ímpar. Ele busca lançará uma exceção caso o valor passado pelo usuário não seja um número inteiro.



Perceba que **‘loop do’** substitui ‘begin’, pois ali já está um bloco, não necessitando do ‘begin’ para estruturá-lo.

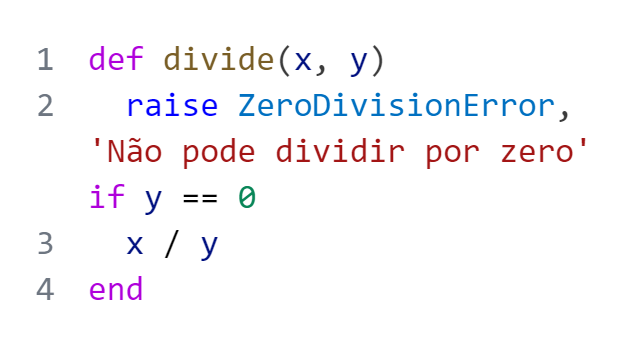
## ‘raise’

É usado para lançar uma exceção explicitamente, inclusive com uma mensagem personalizada opcional.

A sintaxe é a seguinte:



Exemplo:



A mensagem não deve conter acentos, ou não será exibida.

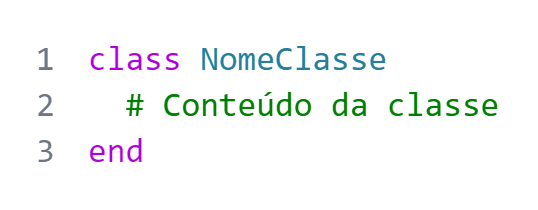
Caso **‘raise’** seja escrito sozinho, irá relançar a exceção capturada.

# Classes

Ruby é uma linguagem de programação orientada a objetos, o que significa que ela manipula objetos.

Uma classe é construída a partir da palavra chave **‘class’**, e após isso, o nome da classe.

A sintaxe é a seguinte:



## Método ‘initialize’

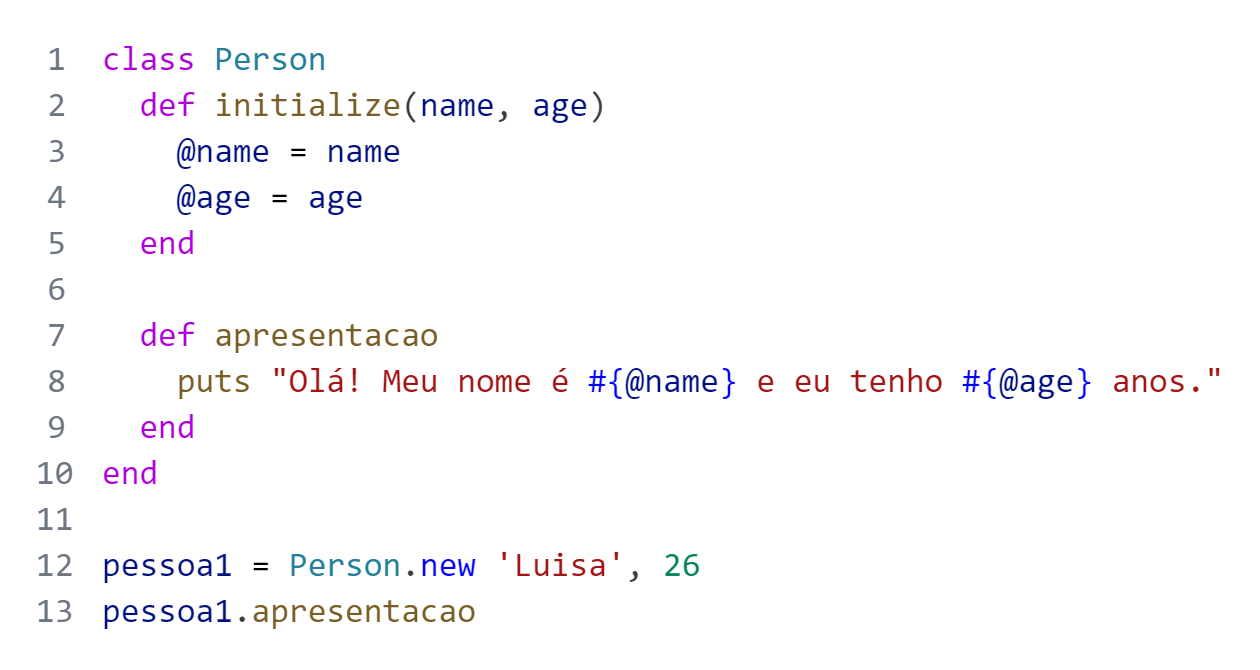
Este método é executando sempre que uma nova instância de uma classe for criada usando o método **‘new’**. Geralmente, é onde são definidas as variáveis de instância. O método **‘initialize’** é opcional.

Os parâmetros deste método devem ser recebidos assim que criada a uma nova instância desta classe.

### Variáveis de Instância

Variáveis de instância são variáveis quer pertencem a uma instância específica de uma classe, ou seja, podem existir dois objetos com a mesma classe, porém com variáveis de instância com valores diferentes.

Variáveis de instância são declaradas com **‘@’** antes do nome da variável.



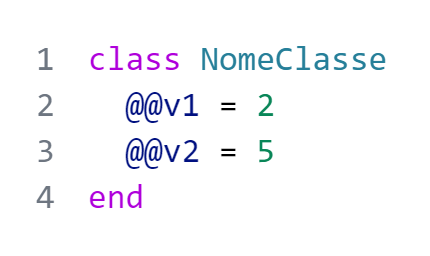
A função ‘apresentacao’ utiliza essas variáveis de instância para mostrar uma mensagem no console.

A variável **‘@name’** é uma variável de instância que recebe o atributo **‘name’** passado pelo usuário, no caso, Luisa.

## Variáveis de Classe

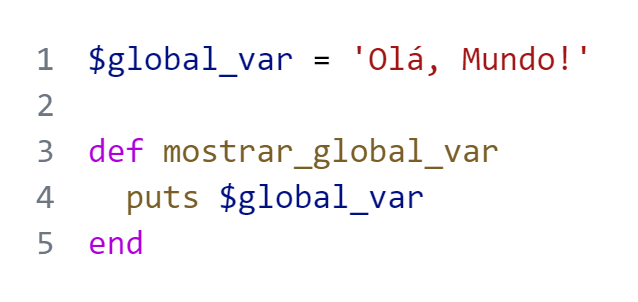
São variáveis que, ao definir ou alterar seu valor em uma instância, altera também em todas as outras. Ou seja, em todas as instâncias as variáveis de classe possuirão o mesmo valor.

Essas variáveis devem ser escritas com **‘@@’** antes de seu nome.



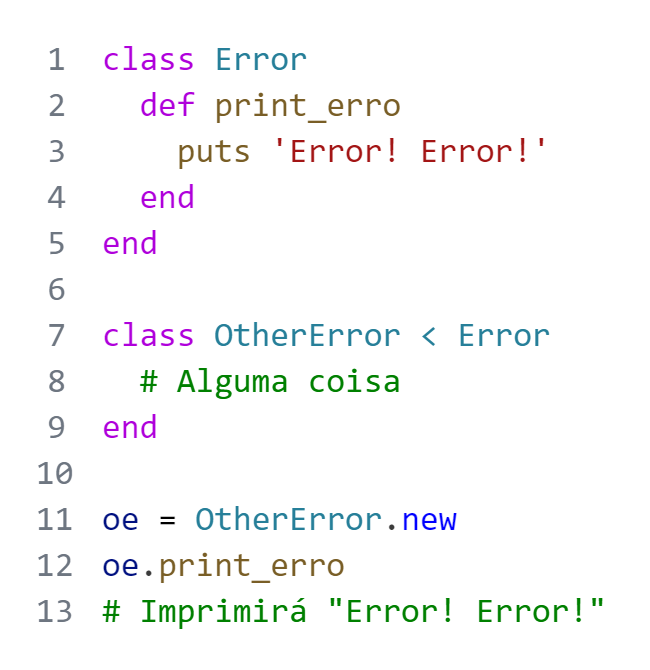
## Variáveis Globais

São variáveis que possuem o escopo global, ou seja, podem ser usadas e alteradas em qualquer local do código. Elas devem ser escritas com **‘$’** antes de seu nome.



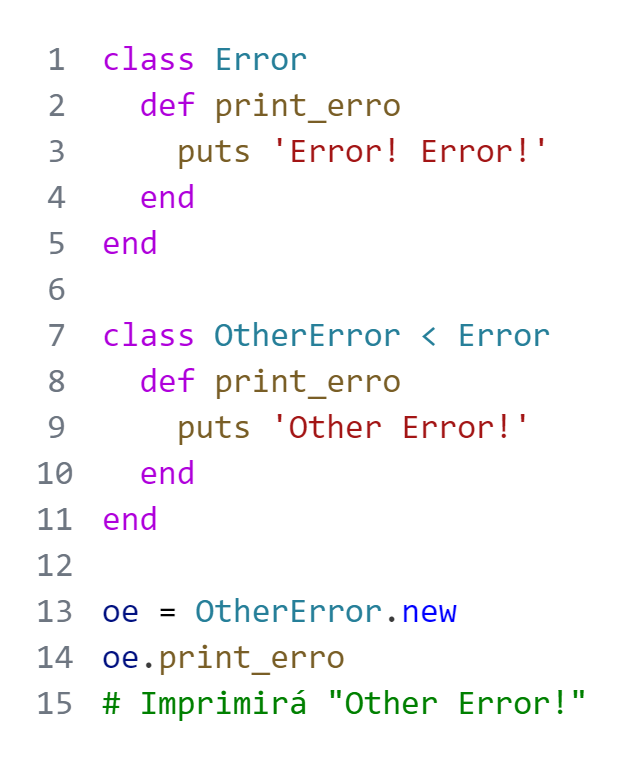
## *Inheritance* (Herança)

Herança é o processo pelo qual uma classe assume os atributos e métodos de uma outra classe. Essa relação de classes é feita por meio do caractere **‘<’**, onde o nome da classe quer herdará vem antes e o nome da classe pai virá depois.



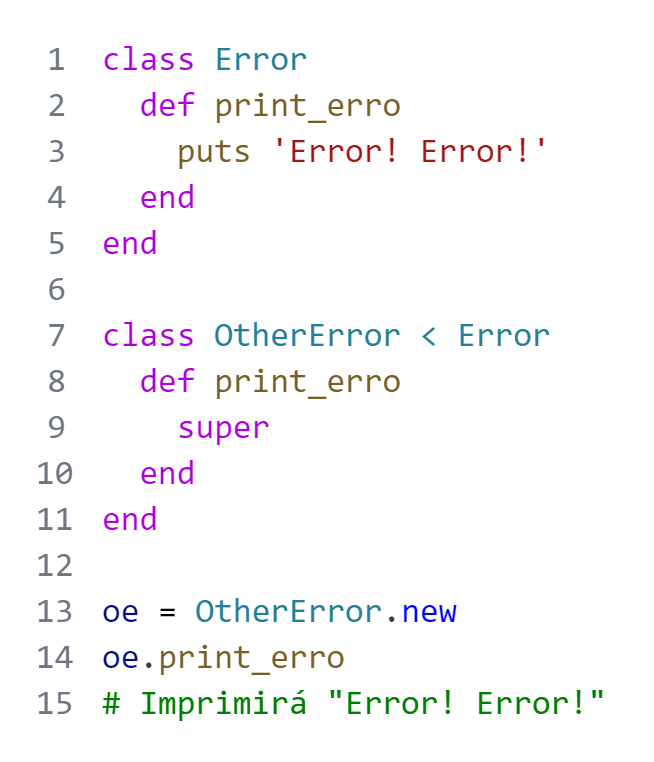
### Substituição de Métodos

É possível substituir métodos da classe pai para a classe que está herdando. Para isso, é só explicitamente recriar o método.



### Executar Versão Original do Método

Com o método **‘super’**, é possível executar a versão original do método com o mesmo nome do que está substituindo.



Caso necessário, podem ser passados parâmetros para **‘super’**.

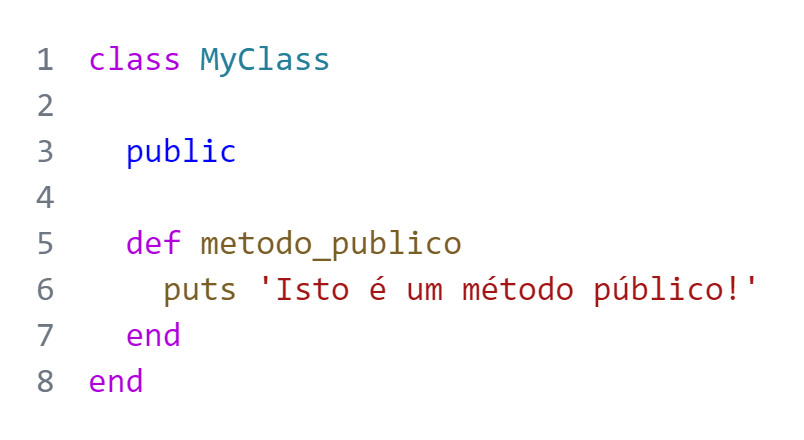
## Restrições a Métodos

### *Public Methods* (Métodos Públicos)

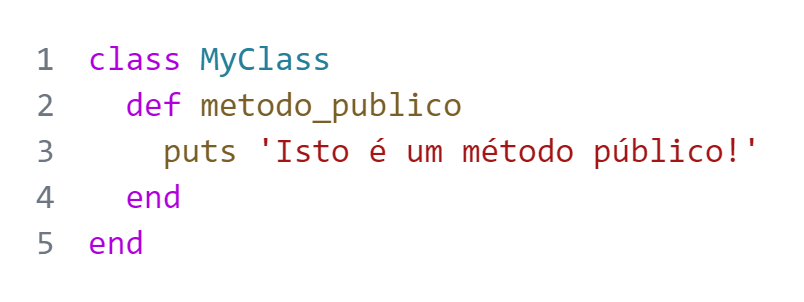
Não há restrições quanto aos métodos públicos. Eles podem ser chamados de dentro ou de fora da classe em que foram definidos.

Qualquer objeto ou instância da classe possuem permissão para acessar métodos públicos definidos a classe.

Ao definir um método, automaticamente ele se torna público ao menos que definido manualmente para privado ou protegido. Mesmo assim você pode explicitamente dizer que é um método público, definindo **‘public’** uma linha antes de definir o método.



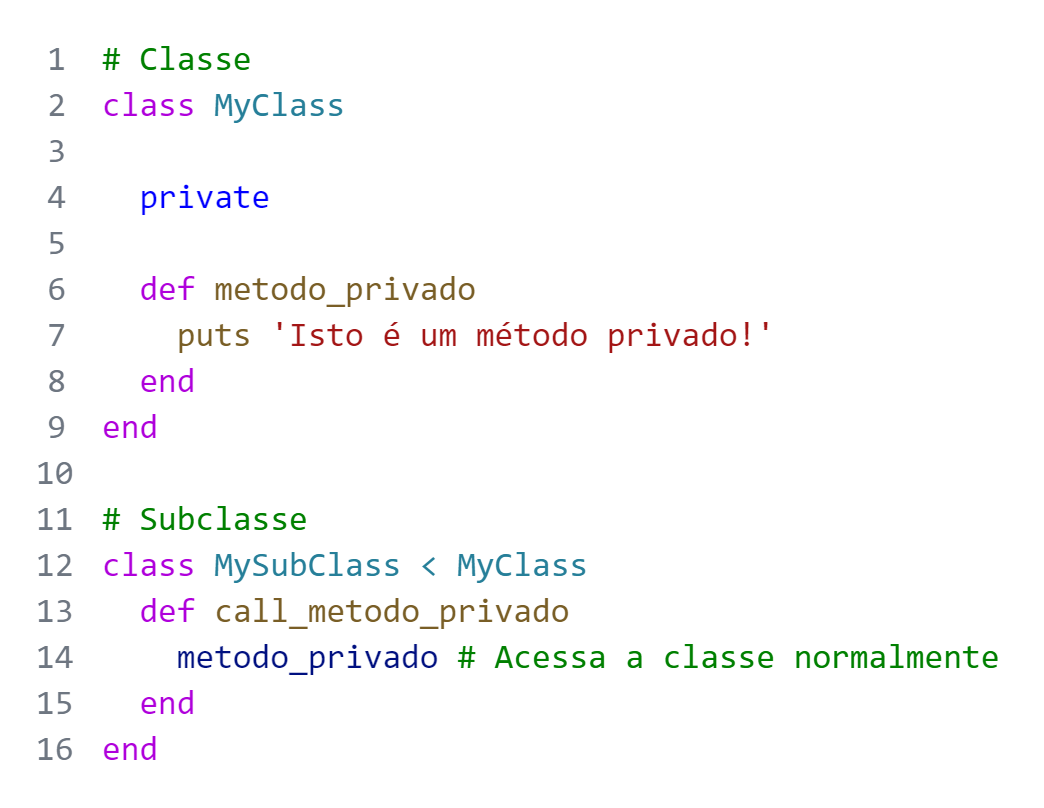
Como dito anteriormente, ao definir um método em uma classe, ele é automaticamente considera público, então o mesmo exemplo acima poderia ser:



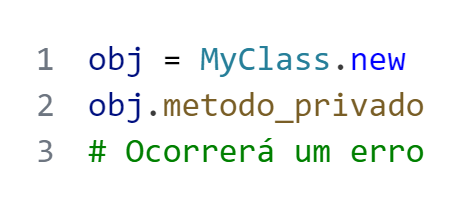
### *Private Methods* (Métodos Privados)

Os métodos privados só podem ser chamados de dentro da própria classe em que foi definido ou em subclasses.

Um método se torna privado ao definir **‘private’** uma linha antes de definir o método.



Considerando a mesma classe acima, veremos um erro chamando o método privado por uma instância da classe acima.



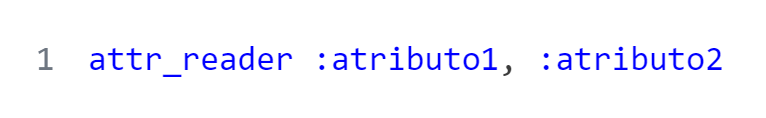
## Métodos de Atributo

Os métodos de atributo são usados para definir automaticamente atributos em uma classe, criando **getters** (métodos de leitura) e **setters** (métodos de escrita) para esses atributos.

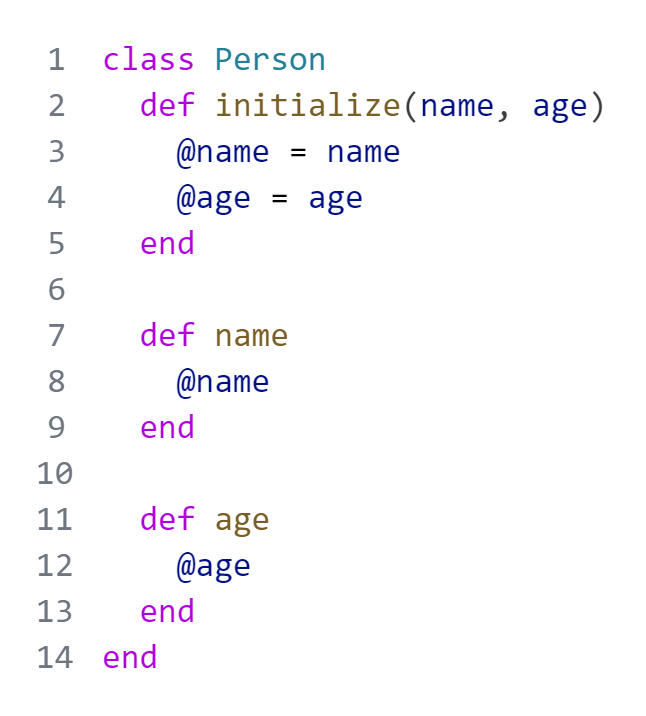
### ‘attr\_reader’

Define automaticamente um método para um ou mais atributos. Os atributos devem ser passados em forma de símbolo.

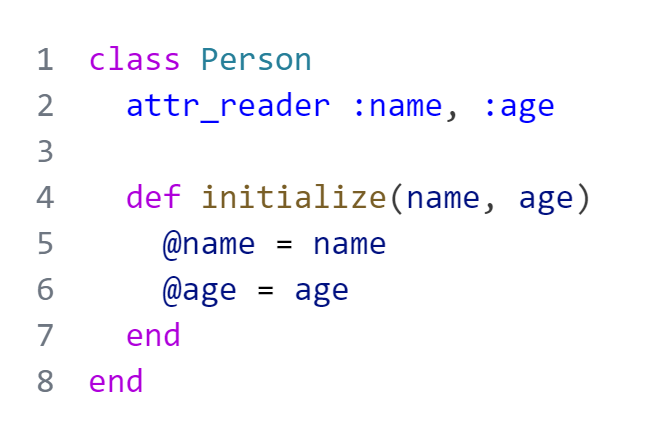
A sintaxe é a seguinte:



Então, ao invés de:



Podemos fazer assim:

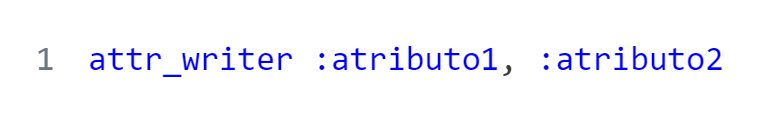


Desta forma, são criados automaticamente os métodos **getters** para ‘name’ e ‘age’ por meio do método **‘attr\_reader’**.

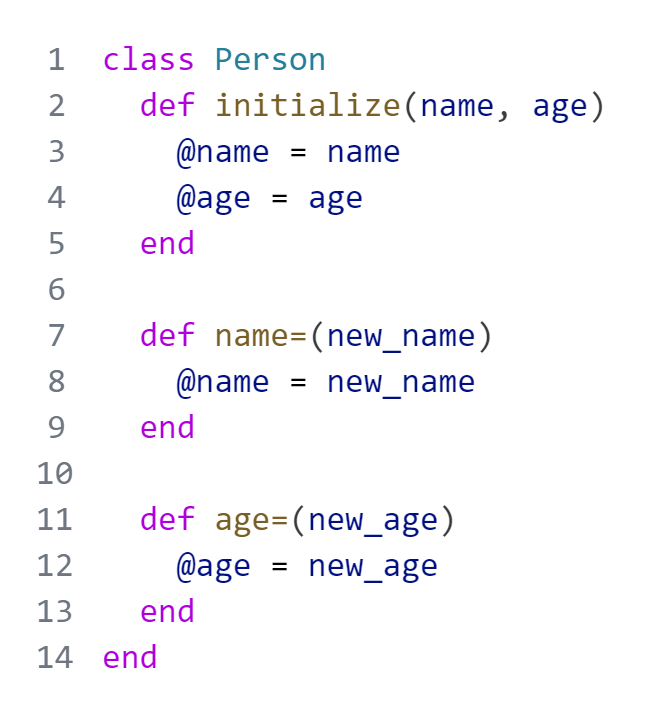
### ‘attr\_writer’

Define automaticamente um método para redefinição de um atributo.

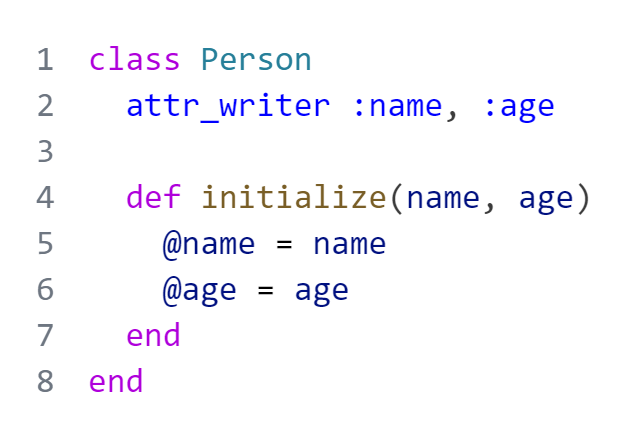
A sintaxe é a seguinte:



Então, ao invés de:



Ficaria:



**‘=’ ao nome do método**: É comum que métodos de definição como os gerados por ‘attr\_writer’ possuam o operador de atribuição ao final. Isso indica melhor a sua função, além de facilitar a digitação.

### ‘attr\_accessor’

Este método é junção dos dois anteriores (‘attr\_reader’ e ‘attr\_accessor’), ou seja, ele cria automaticamente os métodos getters e setters.

