
# Aula 13

## Funções

*Prof. Jalerson Lima*

## Apresentação

Nessa aula iremos estudar as funções em Ruby, que são estruturas capazes de encapsular trechos de código, permitindo que eles sejam chamados quando forem necessários. Isso evita a repetição de código e facilita a manutenção de software.

## Objetivos

1. Compreender a utilidade das funções;
2. Aprender como definir e chamar funções em Ruby;
3. Compreender como passar parâmetros e retornar valores;
4. Exercitar os conhecimentos através de atividades práticas.

# 1. Introdução

Imagine um sistema interno administrativo e financeiro de um grande banco internacional que possui 100.000.000 (cem milhões) de linhas de código. Esse é um sistema realmente grande e complexo. Agora imagine que, nesse sistema, em vários locais é necessário calcular a média aritmética, e para fazer esse cálculo, foi usado o trecho de código ilustrado no Exemplo de código 1.

```
media = (valor1 + valor2) / 2
```

Exemplo de código 1 – Cálculo da média aritmética

Por ser necessário calcular a média em vários locais, essa linha de código se repetiu várias vezes em meio a cem milhões de linhas de código do sistema. Agora imagine que ocorreu uma mudança na legislação financeira e o cálculo da média deve ser alterado para média ponderada (com pesos). E agora? Precisaremos localizar e alterar todos os locais em que o cálculo da média é feito, e essa não será uma tarefa simples, pois esse cálculo é feito em vários locais dentre cem milhões de linhas de código.

As boas práticas de programação nos orientam a não repetir trechos de código várias vezes no sistema, e o motivo você já pode imaginar: caso esse trecho de código precise ser alterado, essa alteração precisará ser feita em vários locais, várias vezes, gastando tempo e, caso a alteração não seja feita num local onde seria necessário, o software estará propenso a apresentar problemas.

Mas como resolvemos esse tipo de situação? Como evitar que trechos de código sejam repetidos várias vezes em um sistema tão grande e complexo? Uma das soluções disponíveis é o uso de funções. Uma função, também conhecida como rotina, procedimento ou método, é uma estrutura capaz de encapsular (“guardar”) um trecho de código, que pode ser executado (chamado) quando necessário.

Portanto, ao encapsular o trecho de código dentro de uma função, podemos chamar essa função nos locais onde o código ficaria, permitindo que ele seja executado quando necessário. Dessa forma, o código fica isolado num único local (na função) mas ele pode ser chamado (executado) quando for necessário. Qualquer alteração no código só precisará ser realizada uma única vez, ou seja, dentro da função.

Antes de começarmos a discutir sobre funções em Ruby, precisamos deixar mais claro as diferenças entre funções, rotinas, procedimentos e métodos. Quando se trata de programação, funções e rotinas têm o mesmo significado, que é esse apresentado anteriormente. Algumas pessoas defendem que procedimentos também tem o mesmo significado, contudo, outras defendem que funções precisam retornar (dar como resultado) um valor, enquanto os procedimentos não têm retorno. Os métodos são funções presentes dentro de classes e, portanto, estão associadas a objetos. Iremos discutir os métodos em profundidade quando começarmos a estudar Programação Orientada a Objetos.

## 2. Funções em Ruby

O Exemplo de código 2 ilustra uma sintaxe para construção de uma função em Ruby.

```
1  def nome_da_funcao<(parâmetros)>
2    // Corpo da função
3  end
```

Exemplo de código 2 – Sintaxe das funções em Ruby

Para criarmos nossa própria função, iniciamos com a palavra-chave `def` seguido do nome da função (linha 1). Os nomes das funções seguem as mesmas regras de nomenclatura das variáveis. As funções podem receber parâmetros, que são dados que podem ser úteis para as funções. Discutiremos melhor sobre os parâmetros em breve. O corpo da função, que é o código que será executado quando a função for chamada, fica entre o `def` e o `end` (linha 2).

Vamos criar nossa primeira função: `ola_mundo`. Essa função simplesmente irá mostrar na tela a *String* "Olá mundo!". Confira como construir essa função no Exemplo de código 3.

```
1  def ola_mundo
2    puts "Olá mundo!"
3  end
```

Exemplo de código 3 – Construindo nossa primeira função

Observe que definimos o nome da função (`ola_mundo`) logo após o `def` (linha 1). O corpo da função contém uma única linha de código (`puts "Olá mundo!"`), que será executada quando a função for chamada. Mas como chamar uma função?

Para chamar uma função e, portanto, executar o código dentro dela, basta usar o nome da função (linha 5 do Exemplo de código 4). Após definir a função, basta usar o nome da função para chamá-la, conforme ilustra o Exemplo de código 4. Lembrando que o `gets` (linha 6) só é necessário para que a janela não feche automaticamente após a execução do código.

```
1  def ola_mundo
2    puts "Olá mundo!"
3  end
4
5  ola_mundo
6  gets
```

Exemplo de código 4 – Definindo e chamando uma função

### Atividade 13.1

Crie um *script* em Ruby, defina e chame a função `ola_mundo` conforme ilustrado no Exemplo de código 4.

## 2.1 Passagem de parâmetros

Parâmetros são dados (variáveis) que podem ser passados para as funções executarem suas tarefas. Imagine, por exemplo, uma função que calcula a média aritmética de dois números. Para que essa função possa fazer isso, ela precisa receber, através de parâmetros, os dois números aos quais se deseja calcular a média aritmética.

Observe o Exemplo de código 5, que ilustra a sintaxe de uma função que recebe parâmetros.

```
1  def nome_da_funcao(parametro1, parametro2, parametro3, ...)
2    // Corpo da função
3  end
```

Exemplo de código 5 – Sintaxe da função com parâmetros

Os parâmetros são definidos entre parênteses logo após o nome da função. Os parâmetros funcionam como variáveis que só existem dentro das funções, portanto os nomes dos parâmetros devem obedecer às regras de nomenclatura de variáveis.

Observe o Exemplo de código 6, que ilustra a definição da função `mostrar_mensagem` e que recebe um parâmetro chamado `mensagem`. O objetivo dessa função, conforme sugere o próprio nome, é mostrar uma mensagem na tela do usuário, da mesma forma como funciona a função `puts`. A mensagem a ser exibida é passada através do parâmetro `mensagem`.

```
1  def mostrar_mensagem(mensagem)
2    puts mensagem
3  end
```

Exemplo de código 6 – Exemplo de função que recebe parâmetro

Uma vez definida a função `mostrar_mensagem`, podemos chamá-la conforme ilustra o Exemplo de código 7.

```
1  def mostrar_mensagem(mensagem)
2    puts mensagem
3  end
4
5  mostrar_mensagem("Olá mundo!")
6  gets
```

Exemplo de código 7 – Definição e chamada da função mostrar_mensagem

Observe, na linha 5 do Exemplo de código 7, que chamamos a função `mostrar_mensagem` e passamos o valor `"Olá mundo!"` entre parênteses. Esse valor será assumido pelo parâmetro `mensagem` na função. Contudo, em Ruby, podemos omitir os parênteses na passagem de parâmetros, portanto outra forma de chamar a função `mostrar_mensagem` é aquela ilustrada no Exemplo de código 8.

```
1  mostrar_mensagem "Olá mundo!"
```

Exemplo de código 8 – Passagem de parâmetros sem parênteses

Lembra-se do `puts`? Pois bem, o `puts` é uma função também. Portanto você pode chamá-lo com parênteses: `puts("Olá mundo!")`, ou sem parênteses: `puts "Olá mundo!"`.

O Exemplo de código 9 ilustra a definição e a chamada da função `media`, que recebe dois números como parâmetros (`n1` e `n2`), calcula e mostra na tela do usuário a média aritmética entre esses dois números. Observe nessa função, que quando desejamos que a função receba mais de um parâmetro, separamos os parâmetros com vírgula.

```
1  def media(n1, n2)
2    puts (n1+n2)/2
3  end
4
5  media(2, 4)
6  gets
```

Exemplo de código 9 – Função que calcula a média aritmética entre dois números

Observe o Exemplo de código 10, que ilustra a função `media` apresentada anteriormente. Contudo, dessa vez, os valores passados como parâmetros para essa função são variáveis que contém valores informados pelo usuário (`numero1` e `numero2`).

```
1  def media(n1, n2)
2    puts (n1+n2)/2
3  end
4
5  puts "Digite um número: "
6  numero1 = gets.chomp.to_i
7  puts "Digite outro número: "
8  numero2 = gets.chomp.to_i
9
10 media(numero1, numero2)
11 gets
```

Exemplo de código 10 – Passagem de variáveis como parâmetros para funções

Observe, entre as linhas 5 e 8, que foram lidos valores inteiros para as variáveis `numero1` e `numero2`, e depois essas variáveis foram passadas como parâmetros para a função `media` (linha 10).

## Atividade 13.2

Crie um *script* em Ruby, defina a função `media`, que deve receber três parâmetros (n1, n2 e n3), calcular e mostrar na tela a média aritmética desses três valores.

### 2.1.1 Valores padrão

Os parâmetros de funções podem ter valores padrão. Os valores padrão serão assumidos pelos parâmetros caso nenhum valor seja atribuído a eles. Observe o Exemplo de código 11, que ilustra como definir valores padrão para parâmetros.

```
1  def nome_da_funcao(param1 = valor1, param2 = valor2, ...)
2    // Corpo da função
3  end
```

Exemplo de código 11 – Parâmetros com valores padrão

Os valores padrão são definidos após o sinal de igual, após o nome de cada parâmetro. Vale salientar que os valores padrão são opcionais, portanto qualquer parâmetro pode ou não ter valores padrão. O Exemplo de código 12 ilustra a função `dividir`, que recebe dois parâmetros numéricos (`dividendo` e `divisor`), sendo que o segundo parâmetro (`divisor`) possui um valor padrão (`2`). Caso nenhum valor seja informado para o parâmetro `divisor`, ele irá assumir o valor `2`.

```
1  def dividir(dividendo, divisor = 2)
2    puts dividendo / divisor
3  end
4
5  dividir(4)
6  gets
```

Exemplo de código 12 – Exemplo de parâmetro com valor padrão

Observe, na linha 5, que chamamos a função `dividir` passando apenas um valor (`4`), que será assumido pelo parâmetro `dividendo`, enquanto o parâmetro `divisor` irá assumir o valor padrão (`2`).

## Atividade 13.3

Crie um *script* em Ruby, defina e chame a função `dividir`, conforme ilustrado no Exemplo de código 12.

### 2.1.2 Quantidade variável de parâmetros

Em Ruby, funções podem receber uma quantidade variável de parâmetros. Para isso, definimos um único parâmetro com um asterisco na frente, conforme ilustra o Exemplo de código 13.

```
1  def nome_da_funcao(*param)
2    // Corpo da função
3  end
```

Exemplo de código 13 – Função com quantidade variável de parâmetros

Nesse caso, `param` será um *array* que irá conter todos os valores passados como parâmetro. Observe o Exemplo de código 14, que ilustra um exemplo de função que recebe uma quantidade variável de parâmetros.

```
1  def somatorio(*numeros)
2    somatorio = 0
3    for numero in numeros
4      somatorio = somatorio + numero
5    end
6    puts somatorio
7  end
8
9  somatorio(1, 2, 3)
10 somatorio(10, 20, 30, 40, 50)
11 gets
```

Exemplo de código 14 – Exemplo de função com quantidade variável de parâmetros

A função `somatario` irá receber um conjunto de parâmetros numéricos e irá realizar o somatório desses números. Observe, na linha 9, que foram passados apenas três valores como parâmetros para essa função (`1`, `2`, e `3`), enquanto na linha 10, foram passados cinco valores (`10`, `20`, `30`, `40` e `50`).

## Atividade 13.4

Crie um *script* em Ruby e defina a função `maior`. A função deve receber uma quantidade variável de parâmetros e exibir na tela o maior valor recebido.

## 2.2 Retorno

O retorno de uma função é aquilo que ela dá como resultado. Uma função que calcula o somatório de um conjunto de números dará como retorno o somatório. Uma função que calcula a média de dois números, irá retornar a média. Uma função que encontra o maior valor em um *array*, irá retornar o maior valor encontrado.

Os exemplos de funções que mostramos anteriormente nessa aula estavam mostrando na tela o resultado da função. Contudo essa é uma prática não indicada, pois, de forma geral, o resultado de uma função deve ser retornado, e não mostrado na tela. Mas por que? Funções normalmente tem tarefas muito específicas que produzem algum resultado que será utilizado de alguma forma por quem chamou a função. Portanto o que será feito com esse resultado é de responsabilidade de quem chamou a função, e não da função em si. Quando a função exibe o resultado na tela, ela está fazendo algo que, de forma geral, não é de sua responsabilidade, a não ser que exibir o dado na tela seja objetivo da função. Portanto sempre prefira retornar o resultado da função ao invés de exibi-lo na tela.

Para retornar um valor ou uma expressão numa função, basta usar a palavra-chave `return` seguido daquilo que deve ser retornado. Observe, no Exemplo de código 15, que a função `somar` está retornando a soma entre `n1` e `n2`, que são parâmetros passados para a função.

```
1  def somar(n1, n2)
2    return n1 + n2
3  end
4
5  puts somar(1, 2)
6  gets
```

Exemplo de código 15 – Exemplo de função com retorno

Na linha 5 usamos a função `somar` e o retorno dela é passado como parâmetro para função `puts`, que irá exibir o retorno da função `somar` na tela.

O Exemplo de código 16 ilustra a função `somatorio`, apresentada anteriormente, reescrita para retornar o resultado ao invés de exibir na tela. Perceba que a única mudança na função é a substituição da função `puts` pelo `return` na linha 6. Fora da função, foram alteradas as linhas 9 e 10, para inclusão da função `puts` para exibir na tela o retorno da função `somatorio`.

```
1  def somatorio(*numeros)
2    somatorio = 0
3    for numero in numeros
4      somatorio = somatorio + numero
5    end
6    return somatorio
7  end
8
9  puts somatorio(1, 2, 3)
10 puts somatorio(10, 20, 30, 40, 50)
11 gets
```

Exemplo de código 16 – Exemplo da função somatório com retorno

Quando você criar uma função que não tiver `return`, a função sempre irá retornar o resultado da última expressão no corpo da função. Observe um exemplo disso no Exemplo de código 17.

```
1  def potencia(base, expoente)
2    base ** expoente
3  end
4
5  puts potencia(4, 2)
6  gets
```

Exemplo de código 17 – Retornando valores sem o *return*

Ao executar o Exemplo de código 17, a função `potencia` irá retornar o resultado de `base ** potencia` mesmo sem usarmos a palavra-chave `return`. Isso ocorre porque `base ** potencia` é a última (e única) expressão da função `potencia`, portanto será retornado pela função.

## Atividade 13.5

Crie um *script* em Ruby e defina a função `menor`. A função deve receber uma quantidade variável de parâmetros e retornar o menor valor informado.

## Atividade 13.6

a) Crie uma função que receba um número inteiro como parâmetro e mostre a tabuada (apenas de multiplicação) desse número;
b) Crie uma função que receba um número inteiro como parâmetro e mostre todos os números pares entre 1 e esse número;
c) Crie uma função que receba um número inteiro como parâmetro e retorne o somatório dos números pares entre 1 e o número passado como parâmetro;
d) Crie uma função que receba um número inteiro como parâmetro e mostre todos os números entre 1 e esse número que são divisíveis por 3 ou por 5;
e) Crie uma função que recebe dois números inteiros como parâmetro, por exemplo X e Y. A função deve retornar a potência entre X e Y, ou seja, $X^Y$ (X elevado a Y);
f) Crie uma função que recebe um número inteiro como parâmetro e retorna o fatorial desse número;
g) Crie uma função que recebe um número inteiro como parâmetro e retorna um valor lógico. A função deve retornar VERDADEIRO caso o número seja primo, e FALSO caso contrário;
h) Crie uma função que recebe um número inteiro como parâmetro e retorna um valor lógico. A função deve retornar VERDADEIRO caso o número seja triangular, e FALSO caso contrário. Um número é triangular quando é resultado do produto de três números consecutivos. Exemplo: 24 é triangular pois 2 x 3 x 4 = 24;
i) Crie uma função que recebe dois números inteiros como parâmetro e retorna o Mínimo Múltiplo Comum (MMC) entre esses números;
j) Crie uma função que recebe dois números inteiros como parâmetro e retorna o Máximo Divisor Comum (MDC) entre esses dois números;
k) Crie uma função que recebe três números inteiros como parâmetros e retorna um valor booleano. A função deve retornar VERDADEIRO quando os três valores puderem ser lados de um triângulo, e FALSO caso contrário. Para saber se três valores podem ser lados de um triângulo, use a seguinte propriedade: a soma de dois lados quaisquer tem que ser maior do que o terceiro lado;
l) Crie uma função que recebe três números inteiros como parâmetros e retorna uma *String*. A função deve retornar "Triângulo Equilátero" quando os valores passados como parâmetro formarem um triângulo equilátero (os três lados iguais); deve retornar "Triângulo Isósceles" quando os valores passados como parâmetro formarem um triângulo isósceles (apenas dois lados iguais); e deve retornar "Triângulo Escaleno" quando os valores formarem um triângulo escaleno (os três lados diferentes). A função deve retornar "Os valores informados não formam um triângulo" quando os valores passados como parâmetro não formarem um triângulo, conforme explicado na questão anterior;
m) Crie uma função que recebe uma nota de 0 a 100 e mostre o conceito relativo à nota. De 0 a 20 = E; de 20 a 40 = D; de 40 a 60 = C; de 60 a 80 = B; e de 80 a 100 = A;

n) Crie uma função que recebe um número inteiro como parâmetro, representando um ano, e retorna um valor lógico. A função deve retornar VERDADEIRO quando o ano for bissexto, e FALSO caso contrário. Um ano é bissexto se ele for divisível por 400; ou se ele for divisível por 4 e não por 100;

o) Crie uma função que recebe um valor real como parâmetro, representando o salário de uma pessoa. A função deve calcular e retornar quanto essa pessoa pagará de imposto de renda. Se o salário for menor do que R$ 1.000,00, não há imposto; se for entre R$ 1.000,00 e R$ 2.200,00, o imposto é de 13% do salário; se for maior do que R$ 2.200,00, o imposto é de 22% do salário;

p) Crie uma função que recebe três números inteiros como parâmetro, representando o dia, o mês e um ano. A função retornará VERDADEIRO quando o dia, o mês e o ano formarem uma data válida, e FALSO caso contrário. Uma data é válida quando o dia estiver entre 1 e 31, o mês estiver entre 1 e 12 e o ano for maior do que zero;

q) Crie uma função que recebe dois valores reais como parâmetro, um representando o peso de uma pessoa, e o outro a altura. A função deve calcular o IMC dessa pessoa com base nos valores passados por parâmetro. Considere que IMC = peso / altura$^2$;

r) Crie uma função que recebe um valor real como parâmetro e mostra uma mensagem na tela. O valor recebido como parâmetro representa o IMC de uma pessoa, e a mensagem é a situação dela conforme o IMC passado como parâmetro.
   - Se o IMC for abaixo de 17 deverá mostrar “Muito abaixo do peso”;
   - Se o IMC for entre 17 e 18,49 deverá mostrar “Abaixo do peso”;
   - Se o IMC for entre 18,5 e 24,99 deverá mostrar “Peso normal”;
   - Se o IMC for entre 25 e 29,99 deverá mostrar “Um pouco acima do peso”;
   - Se o IMC for entre 30 e 34,99 deverá mostrar “Um pouco obeso”;
   - Se o IMC for entre 35 e 39,99 deverá mostrar “Obesidade severa”;
   - Se o IMC for acima de 40 deverá mostrar “Obesidade mórbida”.

## Resumindo

Essa aula apresentou o conceito de funções, sua utilidade, como defini-las e chama-las em Ruby. Além disso, aprendemos como passamos parâmetros para funções, como retornar valores e praticamos os conhecimentos através de atividades práticas.

## Referências


POINT, T. Ruby Tutorial. **Tutorials Point**, 2015. Disponivel em: <http://www.tutorialspoint.com/ruby/>. Acesso em: 12 nov. 2015.

RANGEL, E. **Conhecendo Ruby**. [S.l.]: Leanpub, 2014.

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