

# Estatística aplicada à linguagem de programação Ruby

Jônatas Davi Paganini

fevereiro de 2010

# *1 Introdução*

A eminência da estatística, realmente comprova que ela está na moda. O profissional desta área têm conquistado um espaço cada vez maior. Através das fórmulas, é possível visualizar uma série de informações. Na prática é possível distribuir, classificar e sintetizar dados de diferentes formas e ângulos. Cada disposição dos dados pode revelar diversos tipos de informação, serve ao estatístico, analisar e encontrar os significados e respostas esperados.

Um resultado estatístico é composto por uma série de cálculos e disposições que tornam possível chegar aquele resultado. Quando existe uma quantidade relevante de dados e passos, o trabalho do estatístico se torna mais árduo, se não tornar-se automatizado.

Este trabalho é composto por duas raízes básicas de estudo: A estatística aplicada e programação básica na linguagem Ruby. O primeiro busca explicar as fórmulas e exibir os passos para chegar a um determinado resultado. A segunda tem como objetivo automatizar o processamento das fórmulas e passos realizados anteriormente.

O estudo da estatística é simples e objetivo, cada dado é captado por uma pesquisa com algum objetivo, é executado em um determinado tempo e local. Em outras palavras, uma pesquisa deve ser composta pelos elementos básicos: O quê, quando e quem.

Através do estudo da estatística, exemplo de cálculos práticos, este artigo procura abordar o uso da linguagem de programação Ruby como ferramenta para automatizar os cálculos estatísticos e codificação das fórmulas e decomposição do raciocínio lógico. Através dos exemplos práticos de codificação, cada assunto será codificado, e os novos elementos da linguagem de programação Ruby, serão explicados, após o seu uso, tornando o ensino da programação, uma formalidade do assunto precedido.

Este estudo pretende abordar o conteúdo de forma sucinta, tornado a linguagem de programação apenas uma ferramenta simples para ajudar a automatizar o processo. Desta maneira, é possível experimentar diversas abordagens da linguagem de programação para resolução dos problemas.

## 2 *Estatística descritiva*

A estatística descritiva, é a técnica usada para resumir, comparar, observar e descrever informações relevantes de um ou mais conjuntos de dados. Um conjunto de dados pode ser analisado basicamente em dois passos: organização e método.

Segundo Jairo e Gilberto (1), a estatística descritiva, como o próprio nome sugere, se constitui num conjunto de técnicas que objetivam descrever, analisar e interpretar os dados numéricos de uma população ou amostra.

### 2.1 Distribuição de frequência

#### 2.1.1 Na estatística

Dado uma pesquisa realizada na turma do último ano de sistemas de informação, realizada no ano de 2010, buscando saber mais informações a respeito da faixa etária da turma, foram obtidas as seguintes idades:

23, 31, 31, 21, 31, 26, 31, 22, 22, 24, 31, 21, 22, 20, 22, 31, 21, 20

A partir das idades (dados) coletados acima, serão analisados e classificados os dados. Os dados acima exibidos, estão na forma bruta, ou seja, aleatória e desorganizada.

Em Ruby, podemos representar os mesmos dados brutos através de um vetor de números, e podem ser atribuídos a uma variável chamada **dados\_brutos**.

Listing 2.1: Atribuindo dados brutos a uma variável

```
1 dados_brutos = [23, 31, 31, 21, 31, 26, 31, 22, 22,  
2               24, 31, 21, 22, 20, 22, 31, 21, 20]
```

No exemplo acima, `dados_brutos` é o nome da variável que recebe o vetor de números. O operador `=`(igual) identifica que a variável `dados_brutos` é igual ao vetor declarado posteriormente.

Um vetor ou *Array* é delimitado pelo compilador Ruby pelos caracteres de abertura de colchetes para iniciar os elementos, e fechamento de colchetes para declarar o fim dos elementos.

A quebra de linhas e espaços em branco não fazem diferença alguma para o compilador, os dados acima estão dispostos em duas linhas para ficar mais legível para estudo.

## 2.2 Análise e classificação dos dados

Para entender e analisar os dados com mais facilidade, o primeiro passo é ordenar os elementos por ordem crescente ou decrescente. Neste caso, usaremos ordem crescente.

### 2.2.1 Rol

Este método de organização está dos dados em ordem crescente é também conhecido como Rol. Organizando então o Rol dos dados obtidos na pesquisa acima tem-se:

20, 20, 21, 21, 21, 22, 22, 22, 22, 23, 24, 26, 31, 31, 31, 31, 31, 31

### 2.2.2 Implementando Rol em Ruby

Seguindo a linha do código obtido anteriormente, em Ruby, é possível obter o mesmo resultado facilmente ordenando os valores do vetor. Nativamente, Ruby já traz este método de ordenação e o nome dele é `sort`.

Listing 2.2: Atribuindo dados brutos a uma variável

```
3 rol = dados_brutos.sort
```

Neste caso, analisando o código acima descrito, têm-se uma variável **rol** recebendo o valor do método `sort`, que ordena os **dados\_brutos**.

Inspecionando o resultado da variável **rol**, temos o seguinte vetor:

Listing 2.3: Inspecionando a variável rol

```
4 p rol # [20, 20, 21, 21, 21, 22, 22, 22, 22, 23, ...]
```

Através do método **p** é possível imprimir em tela os valores de um objeto em Ruby. Este método tem o objetivo de ajudar o programador a entender como funciona o código e visualizar as propriedades de uma variável. No exemplo anterior, foi usado para visualizar se a ordem das idades que estão na variável **rol** está correta.

### 2.2.3 Frequência absoluta (fi)

Sabendo que o **rol** já está organizado, é possível inserir as frequências absolutas dos dados. A primeira informação a se extrair do **rol** é a **frequência absoluta**, também caracterizado pela sigla (fi). O **fi** é determinada pelo número de repetições de cada dado da consulta. Neste exemplo, é dada pela quantidade de alunos com a mesma idade.

Desta forma teríamos a seguinte classificação:

20, 20

21, 21, 21

22, 22, 22, 22

23

24

26

31, 31, 31, 31, 31, 31

Conforme os números acima, é possível imaginar que a frequência absoluta dos alunos com 20 anos de idade é 2, assim como, 21 é 3, 26 é 1 e existem 6 alunos com 31 anos.

Entendendo que o objetivo deste cálculo é contar os dados com mesmo valor, já é possível avançar este passo na programação.

### 2.2.4 Implementando a frequência absoluta (fi) em Ruby

Sabendo que a variável **rol** possui as idades ordenadas, existem várias formas de implementar a solução. A primeira forma, aborda a contagem de elementos de cada tipo, e o armazenamento dele é feito através de um **Hash**. Este tipo de elemento, funciona como uma estrutura, que permite adicionar chaves com um determinado valor. Esta chave, pode ser um objeto de qualquer tipo, e também é possível recuperar ou estabelecer um valor para uma chave.

O algoritmo que será criado, consiste em criar um **Hash** ou seja, uma variável que possua várias chaves e valores. E cada chave, será representada pela própria idade, e o valor, será a quantidade de vezes que aquela idade passou está presente no **rol**.

Se não existir um (fi) para aquela idade, então será adicionado o valor 1 para aquela idade. Caso contrário, se encontrar a idade, será adicionado mais 1 na frequência absoluta desta idade.

Listing 2.4: criando o fi

```

5  fi = {}
6  rol.each do |idade|
7    if not fi[idade]
8      fi[idade] = 1
9    else
10     fi[idade] += 1
11   end
12 end

```

O código acima é iniciado pela declaração de uma nova variável, que se chama **fi** e sabe-se que é do tipo **Hash**, pois é inicializada com chaves({}).

Logo na linha 2, é iniciado uma iteração(método **each**) com cada **idade** do **rol**. Esta variável (**idade**) é declarada após o método **each** e reconhecida pelo **bloco** delimitado por **do** e **end**(linhas 6 e 12). As variáveis de bloco, declaradas após **do**(linha 6) só funcionam dentro do bloco(**do..end**) e são interpoladas pelo caracter |.

Para simplificar a explicação, é possível acompanhar na prática como este código funciona. A variável em foco neste momento, é **fi**, ou seja, o **Hash** de **idades** com suas respectivas **quantidades**. Usando o método **p** na variável **fi**, é possível acompanhar passo a passo, as frequências absolutas sendo somadas.

Uma simples inspeção pode ser adicionada, como neste fragmento de código:

Listing 2.5: Inspeccionando a soma dos elementos

```

1  p fi

```

Se o código acima for adicionado após a linha 6 da listagem do fi 2.4, então irá imprimir as seguintes linhas:

Listing 2.6: inspeção da variável fi

```

1  {}
2  {20=>1}
3  {20=>2}
4  {20=>2, 21=>1}
5  {20=>2, 21=>2}
6  {20=>2, 21=>3}
7  {22=>1, 20=>2, 21=>3}

```

```

8 {22=>2, 20=>2, 21=>3}
9 {22=>3, 20=>2, 21=>3}
10 {22=>4, 20=>2, 21=>3}
11 {22=>4, 23=>1, 20=>2, 21=>3}
12 {22=>4, 23=>1, 24=>1, 20=>2, 21=>3}
13 {22=>4, 23=>1, 24=>1, 20=>2, 26=>1, 21=>3}
14 {22=>4, 23=>1, 24=>1, 31=>1, 20=>2, 26=>1, 21=>3}
15 {22=>4, 23=>1, 24=>1, 31=>2, 20=>2, 26=>1, 21=>3}
16 {22=>4, 23=>1, 24=>1, 31=>3, 20=>2, 26=>1, 21=>3}
17 {22=>4, 23=>1, 24=>1, 31=>4, 20=>2, 26=>1, 21=>3}
18 {22=>4, 23=>1, 24=>1, 31=>5, 20=>2, 26=>1, 21=>3}
19 {22=>4, 23=>1, 24=>1, 31=>6, 20=>2, 26=>1, 21=>3}

```

Observe como na primeira iteração o **fi** está vazio, e logo a idade **20** (linha 2) ganha a primeira quantidade **1**. Na segunda vez, o **fi** para idade 20 já existe, então acrescenta-se mais **1** ao **fi** desta **idade**(executa o código do **else**). Na próxima linha o ciclo se inicia novamente com a **idade 21**, fazendo com que a variável **fi** ganhe uma quantidade 1 para a idade 21.

## 2.2.5 Entendendo o funcionamento de um Hash

A classe **Hash** não possui ordenação, por este motivo, a impressão dos elementos acima, não manteve a mesma sequência entre as idades e as quantidades respectivas, mas manteve consistência nos valores.

Este tipo de objeto também pode ser declarado da seguinte forma:

Listing 2.7: Sintaxe de declaração de um **Hash**

```

1 aluno = {
2   :nome => "jonatas",
3   :idade => 23
4 }

```

No código acima, uma variável chamada **aluno** foi declarada, esta variável é um tipo de **Hash**, e é delimitada por abertura de chaves e fechamento de chaves. Os elementos internos como **:nome** e **:idade** são conhecidos como chaves, e **"jonatas"** e 23 são seus respectivos valores. Desta forma é possível acessar os valores através das chaves, e as chaves devem ficar entre colchetes, conforme o exemplo abaixo.

Listing 2.8: Usufruindo dos métodos do **Hash**

```
1 puts aluno[:nome]
2 aluno[:idade] += 10
```

A primeira linha do código anterior imprime o nome do aluno(**jonatas**), enquanto a segunda, adiciona mais 10 anos a chave **:idade**. O mesmo exemplo da segunda linha poderia ser escrito da seguinte forma:

Listing 2.9: Somando 10 anos a chave :idade

```
1 aluno[:idade] = aluno[:idade] + 10
```

Desta forma é possível deduzir que uma atribuição de uma nova chave em um **Hash**, é da mesma forma como a concatenação de um valor.

Listing 2.10: Atribuindo o valor 23 para a chave :idade da variável aluno

```
1 aluno[:idade] = 23
```

## 2.3 Tabelas Estatísticas

Existem várias exigências para a criação de uma tabela estatística, e estas, se encaixam em uma estrutura de:

- cabeçalho
- corpo
- rodapé

A tabela estatística, Com estes três elementos descritos de forma coerente, é possível ler os dados de uma tabela sem esforço.

O primeiro passo, é declarar o cabeçalho, que deve conter informações suficientes para responder **o quê**, **quando** e **onde**. Deve primeiramente descrever **o quê** está demonstrando, deve apresentar uma data ou ano explicando **quando** ocorreu a pesquisa ou fato. E por último, deve apresentar a fonte dos dados, de **onde** eles vieram.



Tabela 1: Pesquisa da idade dos alunos do curso do quarto ano de Sistemas de Informação da Unipar de Francisco Beltrão, no ano de 2010.

Idade	Frequência Absoluta ( <b>fi</b> )
20	2
21	3
22	4
23	1
24	1
26	1
31	6

Fonte: Unipar

Tabela 2: Pesquisa da idade dos alunos do curso do quarto ano de Sistemas de Informação da Unipar de Francisco Beltrão, no ano de 2010.

Idade	Frequência Absoluta ( <b>fi</b> )	Frequência Relativa ( <b>fr</b> )
20	2	2 / 18
21	3	3 / 18
22	4	4 / 18
23	1	1 / 18
24	1	1 / 18
26	1	1 / 18
31	6	6 / 18
$\Sigma$	18	18 / 18 = 100%

Fonte: Unipar

### 2.3.1 Frequência Relativa

Em uma pesquisa, têm-se o número de participantes, por exemplo, na tabela acima consta um total de 18 alunos com idades entre 20 e 31 anos. A frequência relativa é determinada pela razão entre a frequência absoluta (**fi**) e o número total de elementos (como o total de alunos), ou seja, a frequência relativa, é o percentual equivalente na amostra. 2

A frequência relativa de um valor é dada por:

$$fr = \frac{Fi}{n} * 100 \quad (2.1)$$

Sendo assim, é possível transformar os mesmos valores em percentuais, como o exemplo da tabela 3.

### 2.3.2 Implementando a frequência relativa (fr) em Ruby

Sabe-se que a construção da frequência absoluta (**fi**) através da codificação Ruby, está representado através da instância de uma variável chamada **fi** e é do tipo **Hash**. Para **cada dado**

Tabela 3: Pesquisa da idade dos alunos do curso do quarto ano de Sistemas de Informação da Unipar de Francisco Beltrão, no ano de 2010.

Idade	<b>fi</b>	<b>fr</b>	<b>fr %</b>
20	2	2 / 18	11,11
21	3	3 / 18	16,66
22	4	4 / 18	22,22
23	1	1 / 18	5,55
24	1	1 / 18	5,55
26	1	1 / 18	5,55
31	6	6 / 18	16,66
$\Sigma$	18	18 / 18	100,00 %

Fonte: Unipar

da **variável fi** existe uma frequência absoluta, e através do método **each**, é possível iterar sobre cada dado e sua frequência absoluta:

Listing 2.11: Iterando sobre cada dado e frequência absoluta.

```
15 fi.each do |idade, frequencia_absoluta|
```

A linha acima é a declaração de que **cada (each)** dado e frequência absoluta da variável **fi**, será utilizado nas próximas linhas através da variável **idade** que representa a própria idade usada na pesquisa, acompanhado da variável **frequencia\_absoluta**. Desta forma é possível realizar o cálculo da frequência relativa(**fr**) de cada frequência absoluta, usando o mesmo 2.1 cálculo antes representado, mas agora usando as variáveis disponíveis.

Listing 2.12: Iterando sobre cada dado e frequência absoluta.

```
13 fr = {}
14 n = dados_brutos.size
15 fi.each do |idade, frequencia_absoluta|
16   fr[idade] = ((frequencia_absoluta.to_f / n) * 100)
17 end
```

Como a frequência relativa também possui um dado e valor, será usado o mesmo tipo de variável da frequência absoluta, por este motivo é inicializado da mesma forma(linha 13).

A variável **n** recebe o tamanho da amostra. Como a variável **dados\_brutos** é do tipo **Array**, ou seja, é um vetor de elementos, ela possui uma propriedade chamada **size** que traduzindo do inglês para português significa **tamanho**. No caso desta pesquisa o total é **18**, ou seja, a quantidade de alunos pesquisados a respeito de suas idades.

Da mesma forma como é iterado sobre o **rol** para encontrar o **fi**, a frequência relativa (**fr**) de cada **idade** é igual (=) a frequência absoluta(**frequencia\_absoluta**) dividido (/) pela quantidade

Tabela 4: Pesquisa da idade dos alunos do curso do quarto ano de Sistemas de Informação da Unipar de Francisco Beltrão, no ano de 2010.

Idade	<b>fi</b>	<b>fr</b>	<b>fr %</b>	<b>fac</b>
20	2	2 / 18	11,11	2
21	3	3 / 18	16,66	5
22	4	4 / 18	22,22	9
23	1	1 / 18	5,55	10
24	1	1 / 18	5,55	11
26	1	1 / 18	5,55	12
31	6	6 / 18	16,66	8
$\Sigma$	18	18 / 18	100,00 %	

Fonte: Unipar

de alunos da pesquisa (**n**) e multiplicado por 100 ( \* **100** ).

### 2.3.3 Frequência absoluta acumulada

Este dado é composto da soma das frequências absolutas dos valores inferiores ou iguais ao valor dado. 4

### 2.3.4 Frequência absoluta acumulada

Em Ruby, da mesma maneira como é criado a frequência relativa e absoluta, é necessário iterar sobre os elementos e contar as suas frequências absolutas. A diferença é que o valor agora deve ser acumulado, somando a frequência absoluta(**fi**) da **idade** à frequência absoluta acumulada até o item anterior. Quando iniciar, não existindo frequência acumulada anterior, o acumulado inicia com o valor da primeira frequência absoluta.

Desta forma, a obtenção dos frequências absolutas acumuladas das idades é estruturada na seguinte regra:

- Buscar todas as idades únicas em ordem crescente
- Iterar sobre cada uma, pegando sua frequência absoluta
- **Se não** houver frequência **acumulada**

Então o valor **acumulado** é igual a frequência absoluta(**fi**) da **idade**

- **Se não**

O valor acumulado é igual a soma dele mesmo(**acumulado**) **mais** a frequência absoluta da idade(**fi[idade]**).

Com base no algoritmo anterior é possível construir o programa em Ruby.

Listing 2.13: Buscar todas as idades únicas em ordem crescente

```
20 fi.keys.sort.each do |idade|
```

A variável **fi**, como já mencionado anteriormente, é do tipo **Hash**, que é uma variável que permite muitas chaves(**keys**) e valores(**values**). Nesta situação, as chaves (**keys**) são as próprias idades das frequências absolutas(**values**). Como é necessário iterar sobre cada idade de ascendente, para trabalhar com idades únicas é usado:

Listing 2.14: Pegando apenas as idades das frequências absolutas

```
1 p fi.keys
```

O comando acima imprime o conteúdo de uma variável do tipo **Array** semelhante a esta, que representa todas as idades de forma única e desorganizada:

Listing 2.15: Idades das frequências absolutas podem ser usadas com o método de acesso

```
1 [22, 23, 24, 31, 20, 26, 21]
```

O próximo passo é organizar os elementos, como feito nos dados brutos para se transformar em rol. Apenas adicionando o método **sort**.

Listing 2.16: Ordenando as idades das frequências absolutas com o método **sort**

```
1 p fi.keys.sort
```

As idades quando ordenadas da forma acima são representadas:

Listing 2.17: Idades das frequências absolutas ordenadas pelo método **sort**

```
1 [20, 21, 22, 23, 24, 26, 31]
```

Com as variáveis dispostas como no caso anterior, é possível iniciar a iteração sobre cada idade de forma ascendente. Para isso é usado o método **each**. Antes de iniciar o cálculo, é necessário inicializar a variável que irá acumular o valor.

Listing 2.18: Inicializando a variável **acumulado**

```
19 acumulado = nil
```

Quando uma variável é inicializada com **nil**, é equivalente ao negação **false**, desta forma, é possível verificar **se não acumulado** ainda, da mesma maneira que o valor **false**.

Listing 2.19: Verificação da existência de um valor acumulado

```
21  if not acumulado
```

Seguindo a especificação do algoritmo anteriormente abordado 2.3.4, **Se não** houver frequência **acumulada**, Então o valor **acumulado** é igual a frequência absoluta(**fi**) da **idade**

Listing 2.20: Valor acumulado é inicializado com a frequência absoluta da idade inicial

```
22  acumulado = fi [ idade ]
```

Quando a variável acumulada receber o primeiro valor, então nas próximos idades, a frequência absoluta da idade será somada ao valor já existente.

Usando a estrutura condicional **if**, é possível verificar uma condição lógica, a negação desta estrutura é utilizada através do comando **else**, que é permitido apenas dentro de uma declaração **if**. No código que compara a não existência do valor acumulado, a existência do mesmo é garantida no bloco **else**: linha 23.

Listing 2.21: Buscar todas as idades únicas em ordem crescente

```
21  if not acumulado
22      acumulado = fi [ idade ]
23  else
24      acumulado += fi [ idade ]
25  end
```

Dentro do bloco **else**, representado neste caso apenas pela linha 24, é atribuído o valor acumulado a soma dele mesmo com a frequência absoluta existente. O operador **+=** (lê-se "mais igual") é utilizado para fazer esta atribuição. A atribuição desta forma é correspondente à:

Listing 2.22: Operador **+=** escrito de outra maneira

```
1  acumulado = acumulado + fi [ idade ]
```

Seguindo estes passos é possível atribuir o **fac** de cada idade com o valor acumulado:

Listing 2.23: **fac** da idade recebe o valor acumulado

```
26  fac [ idade ] = acumulado
```

O cálculo do **fac** completo, trata-se então por percorrer todos as frequências acumuladas **fi**, acumulando o valor no novo **Hash** atribuído a variável **fac** o valor **acumulado** da frequência

absoluta em ordem crescente.

Listing 2.24: fac da idade recebe o valor acumulado

```
18 fac = {}
19 acumulado = nil
20 fi.keys.sort.each do |idade|
21   if not acumulado
22     acumulado = fi[idade]
23   else
24     acumulado += fi[idade]
25   end
26   fac[idade] = acumulado
27 end
```

O uso da variável **acumulado** atribuído **nil** na linha 19, representa a verificação da inicialização da variável ainda não existente. O valor **nil** trata-se também como um valor **false** permitindo usar a sintaxe **if not acumulado**, que também poderia ser inicializado com 0 e comparado usando **if acumulado == 0**.

Listing 2.25: Usando acumulado com valor inicial 0

```
1 fac = {}
2 acumulado = 0
3 fi.keys.sort.each do |idade|
4   if acumulado == 0
5     acumulado = fi[idade]
6   else
7     acumulado += fi[idade]
8   end
9   fac[idade] = acumulado
10 end
```

A abordagem atribuindo **nil** ao valor **acumulado**, torna-se interessante pela clareza de uma inicialização, e garantia a respeito de uma possível frequência acumulado igual à 0. Em outras palavras, se a frequência da primeira classe fosse 0, então o algoritmo iria falhar. Outra estratégia bastante abordada é o uso de um iterador numérico que permite saber qual é o índice do elemento, podendo assim, acumular a partir do primeiro item.

Considerando que o iterador é inicializado com 0, então é possível comparar ao iterador e

não usar a variável com o valor acumulado.

Listing 2.26: Usando acumulado através do iterador **indice**

```
1 fac = {}
2 acumulado = nil
3 fi.keys.sort.each_with_index do |idade, indice|
4   if indice == 0
5     acumulado = fi[idade]
6   else
7     acumulado += fi[idade]
8   end
9   fac[idade] = acumulado
10 end
```

Usando o método **each\_with\_index** é possível resgatar o valor do iterador através da segunda variável do bloco: **indice**. A comparação acontece da mesma forma, e garante que apenas na primeira interação o valor acumulado será a frequência absoluta inicial.

### 2.3.5 Blocos e escopo de variáveis

Uma observação importante, é que os blocos de código, **do..end** tratam-se de um sub-código, o qual mantém um escopo em suas variáveis. Por este motivo, é necessário que a variável **acumulado** exista antes deste bloco.

Quando a variável é inicializada dentro do bloco, ela perde a referência após o fim do bloco, e na interação seguinte sobre os elementos, torna a não existir, perdendo o valor acumulado anteriormente. Desta forma, é extremamente necessário a declaração da variável **acumulado = nil** antes da interação **each**.

O escopo de variáveis em Ruby é claramente explicado no uso da variável **indice** no exemplo anterior<sup>2.26</sup> em que a variável **indice**, apenas está disponível enquanto iterando sobre os dados, trazendo em seu valor, o **indice** da interação, ou seja, qual é a vez que está iterando.

Os valores dos índices iniciam em zero e vão até o último elemento do vetor, sendo este o número de elementos do vetor subtraído de 1.

Após a linha 10, a variável **indice** deixa de existir, pois saiu do seu escopo de uso. Com isso, sabe-se que quando é necessário utilizar depois de um bloco, a variável deve ser declarada antes, para não perder a referência após o bloco.

### 2.3.6 Criação de métodos

Métodos são ações procedurais com algum objetivo específico. Em ruby, são representados por blocos de código definidos entre palavras **def** e **end**. Os métodos podem receber parâmetros e devolver algum resultado. Por exemplo, se quiser criar um método que faz uma soma, pode-se deduzir um nome para o método como **soma** e receber parâmetros neste método, ou seja, os valores que serão somados.

Listing 2.27: implementando um método de soma

```
1 def soma numero , outro_numero
2   numero + outro_numero
3 end
```

O exemplo anterior mostra a declaração de um método com o nome soma, que recebe dois parâmetros, número e outro número. A declaração de parenteses no nome do método é opcional, assim como no seu uso.

A maioria das linguagem de programação coloca parenteses na identificação de parâmetros, então pode ser escrito assim.

Listing 2.28: implementando um método de soma usando parenteses alternativo

```
1 def soma(numero , outro_numero )
2   numero + outro_numero
3 end
```

Após este código acima ser compilado, é possível executar ele, ou seja, usufruir do método que soma usando a seguinte sintaxe:

Listing 2.29: usufruindo do método de soma

```
1 soma(1 , 2)
2 soma 3 , 4
3 end
```

As duas sintaxes acima podem ser executadas e o método responderá por 3 e 7 na mesma ordem.

O método acima não está encapsulado em uma classe, então pode ser executado apartir do módulo Kernel. Mas também poderia ser declarado dentro de uma classe.



### 2.3.7 Criação de classes

Uma classe é um conjunto de comportamentos que é definido por algum tipo, divisão, qualidades, atributos etc. Essa definição, permite compartilhar comportamentos e definir estes atributos. Por exemplo:

Listing 2.30: Criando uma classe

```
1 class Pessoa
2   def caminhar
3     print "caminhando"
4   end
5 end
```

O método caminhar, é um método de instância e só pode ser executado com a instância de um objeto da classe Pessoa. Um objeto, é representado por a instância de uma classe, e uma classe é a definição do comportamento deste objeto.

Anteriormente, foi utilizado a classe Array, que tem um método **each** que permite iterar sobre **cada** elemento do Array.

Da mesma forma os métodos podem ser definidos e utilizados após a instância da classe. Para criar uma instância de uma nova classe, é necessário utilizar o método **new**.

Listing 2.31: Criando uma classe

```
1 jonatas = Pessoa.new
2 jonatas.caminhar
```

Na primeira linha, uma variável chamada **jonatas** foi declarada, recebendo a nova instância da classe **Pessoa**. Após, foi executado o método caminhar, criado anteriormente na classe Pessoa2.30.

O método caminhar neste caso é representativo, e apenas imprime a frase "**caminhando**".

### 2.3.8 Atributos da classe

Atributos da classe, são características que pertencem aquela classe. Em ruby, é possível definir atributos através do método **attr\_accessor** e eles ser usados dentro dos métodos de instância.

Listing 2.32: Criando uma classe com atributos

```
1 class Pessoa
2   attr_accessor :nome
3   def apresentar_se
4     print "oi, eu sou " + @nome
5   end
6 end
```

A classe acima declara um atributo de acesso chamado nome, com o uso deste método, é possível definir um nome para a instância da pessoa, usando o sinal de atribuição (=).

Listing 2.33: Usando uma classe com métodos e atributos de acesso

```
1 estudante = Pessoa.new
2 estudante.nome = "Jonatas_Davi_Paganini"
3 estudante.apresentar_se
```

A segunda linha pode ser lida como: O **nome** do **estudante** é igual a "Jonatas Davi Paganini" ou **estudante.nome** é igual a "Jonatas Davi Paganini".

# *Referências*

- 1 Simon da Fonseca, Jairo; Martins, Gilberto de Andrade. Curso de Estatística. Sexta edição, São Paulo, SP. Editora ATLAS S.A., 1996.