quero ser_ dev 40+

locawebcompany

Lógica de programação em Ruby

CAMPUS CODE

Antes de começar

Se você está começando sua jornada no mundo da programação é nosso dever avisar que você vai precisar criar um ambiente para programar em seu computador. Instalar linguagens e ferramentas, configurar um editor de código e até mudar de sistema operacional podem fazer parte desse processo, mas não se assuste! O conteúdo desta apostila foi pensado para que você consiga ler e fazer exercícios sem essas barreiras.

Escolhemos o Ruby como a linguagem usada ao longo do texto, afinal, Ruby é conhecida como a linguagem amiga do programador. Com uma sintaxe pouco verbosa e que lembra muito expressões em inglês, é uma ótima porta de entrada para aprender algoritmos e lógica de programação. Por essa razão, sugerimos que procure instalar Ruby no seu computador.

Se fazer essa instalação não for possível, não tem problema. Apenas com um navegador e internet você pode exercitar a maior parte do nosso conteúdo. Os sites TryRuby e Repl.it permitem executar código Ruby direto do seu navegador.

O Ruby foi pensado para desenvolvimento em sistemas baseados em Unix, por essa razão recomendamos instalar sistemas operacionais como: Ubuntu, Fedora, Mint e outras distribuições Linux ou macOS.

Se você usa Windows, recomendamos que você faça *dual boot* na sua máquina para não ter problemas de performance. Outra opção é fazer a instalação através de máquinas virtuais. Como as formas de instalação mudam frequentemente, você pode usar como guia a documentação oficial do sistema. Para iniciantes, recomendamos o Ubuntu e seu guia de instalação disponibilizado pela comunidade. Uma alternativa a isso, mas não muito performática, é o Ruby Installer em ambientes Windows.

No Windows 10 também há a opção de instalar um Terminal Ubuntu para simular um sistema Linux dentro do seu Windows. Você pode ler mais neste tutorial de instalação.

Para instalar o Ruby em sistemas Unix como Ubuntu, você pode usar algo simples como abrir o Terminal e digitar o comando `apt-get install ruby`, mas a maioria dos programadores usa gerenciadores de versão para a instalação do Ruby. Esses gerenciadores permitem instalar e alternar entre versões diferentes da linguagem no mesmo computador. Nossa recomendação é o RVM.

De qualquer forma, o mais importante é você ter o ambiente pronto e se sentir confortável com ele. No site oficial da linguagem você encontra outras opções de instalação, além das duas que recomendamos acima.

Com o Ruby instalado, você vai precisar escrever código em um editor de texto. Diferente dos processadores de texto como o Microsoft Word, os editores permitem que você veja realmente todo o conteúdo dos arquivos que está criando. Em programação, uma vírgula ou um espaço em branco na posição errada pode gerar erros nos programas, por isso é importante ter total controle e percepção do código escrito.

Para os editores, temos duas recomendações gratuitas e bastante adotadas pela comunidade: o Atom e o Visual Studio Code, mas existem muitas alternativas e você pode usar a que achar melhor.

Essa apostila utiliza exemplos de código. Você poderia copiar/colar, no entanto sugerimos que você tente digitar todas as linhas para se habituar ao processo. Isso também vai facilitar a fixação do conteúdo.

Lógica de Programação em Ruby

Introdução

A programação ganhou grande relevância com o crescimento e popularização do uso de computadores, de celulares e da internet. Muitas pessoas se sentiram curiosas em conhecer como aquele site ou jogo de celular funcionam e com isso começaram a procurar conteúdos, cursos e faculdades ligados à tecnologia.

O objetivo deste conteúdo é trazer uma abordagem bastante dinâmica sobre os pilares básicos de programação, mesclando conceitos e prática a todo momento. Vamos tratar do conceito de algoritmo e demonstrar as estruturas básicas que as linguagens de programação oferecem para criarmos programas.

Algoritmos

O que é esse tal de algoritmo? Imagine a seguinte situação: você está indo trabalhar e um desconhecido pergunta onde é a estação de metrô mais próxima. Você pensa onde está agora, onde fica a estação, como pode chegar lá e, em seguida, explica passo a passo o caminho.

Essa situação hipotética acontecia com muito mais frequência antes de termos apps com mapas, no entanto, ela traduz a ideia de algoritmo: uma sequência finita de passos que resolve um problema, como uma receita. Em Ciências da Computação esses passos para resolver problemas podem envolver processamento de dados, cálculos, análises preditivas, entre outros.

Quando escrevemos um programa, precisamos lembrar que não estamos falando com um ser humano – que já sabe que é necessário parar no semáforo se ele estiver vermelho –, mas com um computador, que precisa que todas as situações sejam incluídas de forma clara e sem deixar dúvidas. O que isso quer dizer? Você precisa deixar claro que “enquanto o semáforo estiver vermelho, não ande!”

Dica

Em um vídeo bastante popular, um pai pede aos filhos para descrever todos os passos para fazer um sanduíche. Essa é exatamente a ideia que queremos trazer ao apresentar algoritmos. Você pode assistir ao vídeo em: Como ensinar linguagem de programação para uma criança.

Vamos criar um algoritmo para a seguinte situação: um jogo em que você pergunta para um amigo um número qualquer e responde se ele é par ou ímpar. Que tal descrever os passos para esse algoritmo em detalhes?

- Pedir ao seu amigo que diga um número qualquer;
- Anotar este número num papel;
- Dividir esse número por 2;
- Verificar se o resto dessa divisão é zero;
- Se o resto da divisão por 2 for zero, o número é par;
- Se o resto da divisão não for zero, o número é ímpar;
- Diga a resposta para o seu amigo.

Pronto, escrevemos um algoritmo :)

Provavelmente você pode imaginar os passos acima com mais ou menos detalhes. Poderíamos acrescentar mais passos como: tratar números negativos ou se seu amigo não quiser jogar. Mas também podemos ser mais simplistas e resumir todo processo em 2 ou 3 passos.

Quando pensamos em lógica de programação geralmente devemos usar o primeiro modo: detalhando ao máximo todas possibilidades.

Dica

Para padronizar a escrita de algoritmos podemos usar as pseudolinguagens. Com elas escrevemos um algoritmo de forma mais próxima do esperado por um computador, mas ainda não se trata de uma linguagem de programação em si. Esse código pode ser chamado de pseudocódigo. Um exemplo famoso de pseudolinguagem é o Portugol, bastante utilizado para o ensino de algoritmos em português.

Diferença entre algoritmo e código

Para explicar o caminho para a pessoa que estava perdida, não foi necessário escrever um código, mas você pode fazê-lo. Código nada mais é que a tradução de um

algoritmo (uma sequência de ações) para uma linguagem de programação que o computador entenda.

Terminada a tarefa de escrever os passos em texto simples, precisamos convertê-los para uma linguagem de programação. Vamos usar Ruby para o exemplo, mas, neste momento, não esperamos que você compreenda o código – apesar de ser uma linguagem de alto nível e bastante compreensível para leigos (mas isso é assunto para outro momento).

Nosso programa ficaria assim:

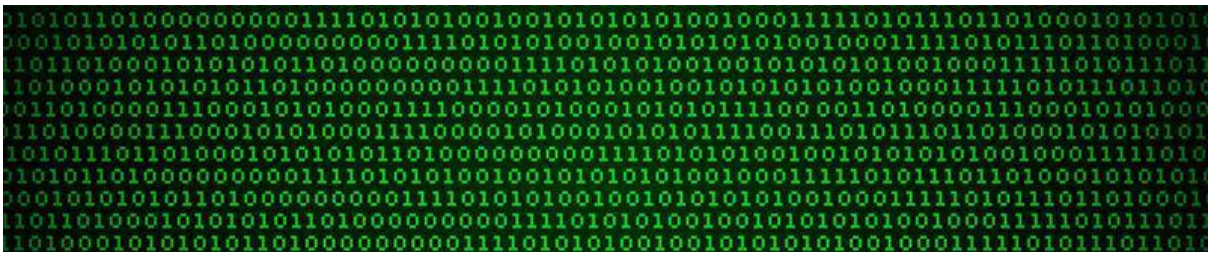
```
# Perguntar um número qualquer para seu amigo;
puts 'Digite um número: '
# Anotar este número num papel
numero = gets
# Dividir esse número por 2
inteiro = numero.to_i()
# Verificar se o resto dessa divisão é zero
resto = inteiro % 2

# Se o resto da divisão por 2 for zero o número é par
if resto == 0
# Diga a resposta para o seu amigo
  puts 'Número é par!'
# Se o resto da divisão não for zero o número é ímpar
else
# Diga a resposta para o seu amigo
  puts 'Número é ímpar!'
end
```

Dica

No código acima, `puts` é um comando para imprimir texto na tela. Você vai utilizar esse comando em exercícios desse material.

Agora que apresentamos o conceito de algoritmos e demonstramos sua diferença para o código executado por um computador, vamos nos aprofundar na criação de código. Para escrever qualquer código precisamos ter uma linguagem de programação que permita enviarmos sequências de comandos para a máquina.

Linguagens de Programação



Ao pensar em código você pode lembrar instantaneamente de sequências de 0 e 1 e de telas pretas como no filme Matrix. No fim das contas o processador do seu computador realmente lê e responde a essas sequências, mas na prática ao longo da breve história da Computação novas linguagens de programação foram criadas para permitir que o processo de escrita de código ficasse cada vez mais natural.

As linguagens de programação modernas permitem escrevermos sequências de comandos muito mais legíveis e fáceis de interpretar, mesmo que seja seu primeiro contato com uma determinada linguagem.

Classificamos linguagens que estão mais próximas do código binário (0 e 1) de linguagens de baixo nível e as linguagens mais legíveis e com mais recursos de linguagens de alto nível. Existem outras classificações e conceitos que podemos aplicar a linguagens, mas, para esse conteúdo, basta sabermos que vamos escrever exemplos e fazer exercícios em Ruby, que é uma linguagem de alto nível.

Cada linguagem de programação oferece recursos e possibilidades diferentes para resolvermos problemas, mas todas elas seguem alguns princípios e têm estruturas mínimas para escrevermos um código. Vamos detalhar esses princípios e componentes básicos que você deve encontrar na maior parte das linguagens.

Entrada e saída

Para resolver problemas através de código vamos precisar receber dados do usuário, processar esses dados e exibir uma resposta na tela diversas vezes. A entrada e saída dessas informações é trivial em qualquer linguagem.

Para imprimir dados na tela com Ruby podemos usar o comando `puts`, como nos exemplos abaixo:

```
puts("Ola")
puts "Mundo"
puts('Hello')
puts 120
```

Se você instalou o Ruby em seu computador, você pode usar o *Interactive Ruby Shell* (IRB) para testar esses blocos de código. Abra o Terminal de seu computador – no Ubuntu

ele fica disponível na busca de aplicativos, por exemplo – e digite `irb` para executar o interpretador de Ruby. Ao executar essas linhas de código você deve ter uma saída similar à da imagem abaixo:

```
campuscode@campuscode:~$ irb
2.6.3 :001 > puts("Ola")
Ola
 => nil
2.6.3 :002 > puts "Mundo"
Mundo
 => nil
2.6.3 :003 > puts('Hello')
Hello
 => nil
2.6.3 :004 > puts 120
120
 => nil
2.6.3 :005 >
```

Dica

Repare que na 1ª e na 3ª linha temos parênteses junto ao `puts` e nas demais só temos o valor que queremos imprimir. Em Ruby, na grande maioria das vezes os parênteses são opcionais. Por questões didáticas vamos sempre usá-los ao longo deste conteúdo.

Receber dados do usuário é tarefa do comando `gets`. Se você tem Ruby instalado, abra o Terminal e rode o IRB. Com o IRB rodando, escreva `gets` e dê `Enter`. O Terminal vai ficar 'congelado' esperando você digitar uma série de letras e números até você confirmar o fim do comando pressionando `Enter`.

Dica

Se você não instalou o Ruby, atenção: no TryRuby não é possível executar o comando `gets`. No Repl.it você não vai encontrar esse obstáculo, então ele acaba sendo o mais recomendado a partir de agora.

Imprimir e receber dados do usuário foram nossos primeiros passos, mas para realizar operações precisamos armazenar esses dados.

Variáveis

Variáveis são pedaços de informação associados a um nome. Na álgebra é comum serem utilizados `x` e `y` para nomear variáveis em equações. Na equação `x + y = 2`, por exemplo, x e y podem ter muitos valores. Nas linguagens de programação as variáveis fazem referência a uma posição na memória do computador onde estão alocadas.

Para declarar uma variável precisamos de um nome, um tipo e um valor. Por exemplo, uma variável para armazenar o nome de uma pessoa, pode se chamar `nome`, seu tipo será um texto (ou *string* como chamamos em programação) e o valor pode ser `'Gustavo'`. Uma outra variável será chamada `nota`, à qual atribuímos neste exemplo a nota `8`. Para declarar uma variável em Ruby vamos usar a seguinte notação:

```
nome = 'Gustavo'
nota = 8
```

Apesar da nossa leitura natural acontecer da esquerda para a direita, ao realizar uma atribuição de valores para variável o Ruby interpreta a linha da direita para a esquerda.

```
valor = 100 - 10
```

No exemplo acima, o cálculo 100 - 10 é realizado primeiro e o resultado é armazenado na variável `valor`.

Em algumas linguagens, uma variável deve ter seu tipo de dados fixado. Em C# ou Java, por exemplo, temos códigos parecidos com o seguinte:

```
string nome = 'Gustavo';
nome = 'João';
nome = 10; // essa atribuição produz erros no código!

# Todas as linguagens de programação possuem formas de adicionar
# comentários no meio do código. Em Ruby, utilizamos #, em
# JavaScript utiliza-se //, por exemplo. Esses comentários são
# ignorados pelo computador na leitura do código e, por isso,
# não interferem na sua execução.
```

A variável `nome` está fixada como uma *string*. Seu valor pode ser substituído por outros textos. Em Ruby isso não acontece. O tipo da variável é flexível:

```
nome = 'Gustavo'
nome = 10 # não faz sentido, mas não temos problemas
```

Nomenclatura

Cada linguagem tem suas próprias regras para nomear variáveis. Além disso, existem boas práticas adotadas pelos desenvolvedores. Em Ruby temos poucas restrições:

- Nomes de variáveis não podem contar espaços;
- Nomes de variáveis não podem começar com números nem conter caracteres especiais como `!`, `&` e letras com acento como `ç` ou `ã`.

Mas, além das regras, cada linguagem acaba adotando algumas convenções. Em Ruby a comunidade adota o seguinte:

- Todas as letras devem ser minúsculas e espaços são substituídos por `_`, uma notação conhecida como *snake_case*;
- Os nomes são dados baseados no conteúdo da variável e sem abreviações.

```
# Exemplos de nomenclatura de variáveis em Ruby
nome_completo = 'Fulano Sicrano'
nota_final = 10
```

Tipos de dados

Tipos de dados indicam a natureza da variável e informam como o programa pretende utilizar essa informação. Tipos de dados ocupam diferentes tamanhos de espaço de memória e permitem diferentes tipos de operações.

Parece complicado e abstrato, mas vamos tentar explicar de forma mais prática: imagine um programa em que você precisa armazenar informações como o nome de um produto, seu preço e a quantidade em estoque. Cada uma dessas informações tem um propósito diferente e permite tomar ações diferentes. Você pode querer imprimir o nome do produto em maiúsculas, verificar se a quantidade em estoque ainda é maior que zero ou aplicar um desconto sobre o preço. Podemos dizer que cada uma dessas 3 informações são de tipos de dados diferentes: um texto para o nome, um número inteiro para o estoque e um número com casas decimais para o preço.

Embora diferentes linguagens de programação possam usar diferentes terminologias, quase todas elas incluem o conceito de tipos de dados. Os tipos mais comuns são: número, *string* e *boolean*. Vamos começar a detalhar esses tipos de dados usando a linguagem Ruby como referência.

Strings

Strings são coleções de caracteres que podem ser letras, números ou pontuação. Uma letra `'a'` ou uma frase como `'Ruby é a linguagem amiga do programador desde 1994'` são exemplos de *strings*.

Em Ruby uma *string* começa com aspas simples ou duplas, seguidas de uma sequência de caracteres e novamente por aspas (simples ou duplas seguindo a primeira) fechando a coleção.

```
uma_string = "Qualquer texto"
outra_string = 'Um texto qualquer'
```

Podemos realizar algumas operações em *strings*. A mais comum é combiná-las.

```
puts('Combinando' + 'Strings')
```

O resultado é `CombinandoStrings`. Esse método é conhecido como **concatenação**. Toda *string* em Ruby possui alguns métodos que serão muito úteis na nossa jornada. Métodos são procedimentos que executam determinadas ações no elemento em que é utilizado. Por exemplo, se precisamos determinar o número de caracteres de uma *string*, podemos usar um **método** chamado `length` ou então podemos colocar todo o texto em maiúscula ou minúscula com `upcase` ou `downcase`.

```
puts( 'Campus Code'.length() )
# => 11
puts( 'Campus Code'.downcase() )
# => "campus code"
puts( 'Campus Code'.upcase() )
# => "CAMPUS CODE"
```

Vamos praticar um pouco esses conceitos? Declare uma variável `nome` e atribua a ela seu primeiro nome como valor. Em seguida, declare a variável `idade` e atribua a ela a sua idade. Em seguida, no editor, escreva `puts 'Meu nome é ' + nome + ' e eu tenho ' + idade + ' anos de idade'` e rode seu código.

```
nome = 'João'
idade = '30'
puts('Meu nome é ' + nome + ' e eu tenho ' + idade + ' anos de
idade')
```

A frase completa deve ser impressa.

Abaixo desse código escreva `puts('Meu nome tem ' + nome.length() + ' caracteres')`. Um erro deve ser impresso! Você consegue entender porquê ele aconteceu?

O método `length` devolve um número, mas quando a frase é impressa, espera-se uma *string*. Por isso o código retorna um erro. Para que ele funcione, você precisa converter o número em uma *string*. Isso pode ser feito com o método `to_s()`. Corrija seu código para que ele fique assim: `puts('Meu nome tem ' + nome.length.to_s() + ' caracteres')` e rode o programa.

Também podemos inserir código Ruby no meio de uma *string* utilizando a **interpolação**. Para isso basta usar a notação `#{seu_codigo}` dentro de uma *string* usando aspas duplas. Por exemplo:

```
nome = 'Joaquim'
puts("Meu nome tem #{nome.length().to_s()} caracteres")
```

Dica

É uma convenção de programadores Ruby usar aspas duplas somente quando houver interpolação e nos outros casos usar aspas simples.

Você se lembra do comando `gets`? Ele recebe um texto digitado por você até que a tecla `Enter` seja pressionada.

Agora você pode usar o `gets` com variáveis:

```
puts('Qual o seu nome?')
nome = gets()
puts("Meu nome é #{nome}")
puts("Meu nome tem #{nome.length().to_s()} caracteres")
```

Se você testar o código acima vai notar que a linha que imprime seu nome foi quebrada em duas linhas e que seu nome parece ter um caracter a mais na conta realizada pelo Ruby.

Isso acontece porque o `gets` em Ruby guarda todo o texto digitado **inclusive** o `Enter`, que é reconhecido pelo código `\n` no final do texto.

```
puts('Qual o seu nome?')
nome = gets()
# => “João\n”
```

Em Ruby este caracter é salvo na *string* junto com o texto. Para limpar a *string* com seu nome desse caractere você pode usar o método `chomp()`, que “come” essa marcação.

```
puts('Qual o seu nome?')
nome = gets().chomp()
# => “João”
```

Agora que o resto do programa funciona perfeitamente vamos falar um pouco das variáveis do tipo número.

Números

Vamos trabalhar basicamente com dois tipos de números quando programamos: *integer* (inteiros) e *floats* (decimais). Ambos podem ser apresentados em seu valor positivo ou negativo. São exemplos de variáveis do tipo *integer*: `1`, `-45`, `0`. Já variáveis do tipo *float* podem ser: `-2.3`, `5.790`, `3.5`. Note que em Ruby o ponto é utilizado como separador decimal (separa a parte inteira da parte decimal de um número).

As operações numéricas mais comuns como `+`, `-`, `/` e `*` são nativas na maioria das linguagens.

```
8 + 1 # 9
7.5 - 2 # 5.5
5/0 # Infinito
```

Também podemos combinar números com variáveis:

```
numero = 8
numero = 4 + 6 #10
```

Se queremos executar operações com uma variável existente podemos fazer isso da seguinte forma:

```
numero = 8
numero = numero + 6
# soma o valor antigo (8) com 6 e guardamos novamente
# na variável que agora vale 14
numero = numero / 2 # 7

numero += 10 # 17
# Executa um código similar ao acima mas é uma forma mais
# elegante e performática de escrever, além do += temos outros
# como: *=, /=, -=, entre outros
```

Dica

Um exemplo comum de código seria receber um texto e transformá-lo em número.

```
puts('Digite um número: ')
numero = gets.chomp()
# => "1"
# "1" não é um número e sim um texto com um número
```

Para fazer isso podemos usar o método `to_i()` ou `to_f()`.

```
puts('Digite um número: ')
numero = gets.to_i()
# => 1
# Agora temos um número
numero = numero + 1
# => 2
```

Além disso, assim como nas *strings*, temos alguns métodos que podem nos ajudar a manipular dados numéricos:

```
10 % 2 # resto da divisão
10.positive?() # informa se o valor é positivo
-1.negative?() # informa se o valor é negativo
1.odd? # informa se o valor é impar
```

Os três últimos métodos apresentados acima retornam `true` (verdadeiro) ou `false` (falso), que são dados do tipo *boolean*.

Booleans

Variáveis do tipo *boolean* podem carregar basicamente dois valores: `true` (verdadeiro) ou `false` (falso). Ruby considera qualquer coisa diferente de `nil` e `false` como `true`. Algumas linguagens de programação não utilizam a nomenclatura `true` ou `false`, preferindo `0` ou `1` por motivos de eficiência de armazenamento na memória.

Mas vamos ao básico primeiro. As expressões abaixo retornam `true` ou `false`:

```
1 < 2 # true
1 > 2 # false
1 == 2 # false
1 != 2 # true
```

Até aqui nada de mais, certo? Mostramos os **operadores** “menor que” `<`, “maior que” `>`, “igual” `==`, “diferente” `!=` e a forma óbvia que o Ruby avalia as expressões com esses operadores.

Atenção ao igual `==`. Somente um sinal `=` é usado para atribuição, como vimos em diversos exemplos até agora.

Outros operadores lógicos são o **E** `&&` e o **OU** `||`. Com eles, você pode combinar múltiplas expressões, veja:

```
1 > 2 || 1 < 2
# => true
1 != 2 && 1 == 2
# => false
```

As expressões acima podem ser lidas da seguinte forma:

1 é maior que 2 OU 1 é menor que 2.

Verdadeiro, pois 1 é menor que dois.

1 é diferente de 2 E 1 é igual a 2.

Falso, pois 1 não é igual a dois.

Os operadores `&&` e `||` operam em “curto circuito”. Isso significa que se em uma expressão com `&&` a primeira avaliação (da esquerda) for falsa, a segunda não será avaliada (por que não há necessidade). Já no caso do `||`, a segunda expressão só será avaliada se a primeira for falsa. Vamos aos exemplos:

```
10 > 5 && 8 < 6 # false
10 == 10 || 10 == "onze" # true
```

Na primeira linha, `10 > 5` é uma expressão verdadeira, mas a segunda é falsa. Como a comparação é feita com o operador `&&` e uma das expressões é falsa, o resultado final é `false`. Já na segunda linha, em que a comparação é feita com `||` apenas uma das expressões precisa ser verdadeira para retornar `true`. Como a primeira expressão (`10 == 10`) é verdadeira, não importa qual é o resultado da segunda expressão, o resultado dessa linha sempre será `true`.

Você pode fazer alguns testes alterando as expressões comparadas e observe o resultado impresso para verificar se compreendeu como funcionam os operadores `&&` e `||`.

Arrays

Uma estrutura muito utilizada quando escrevemos código são os *arrays*. Basicamente são listas ordenadas de elementos. Cada uma das posições é numerada sequencialmente e pode conter algum elemento ou estar vazia. Em Ruby, um *array* é declarado com a notação de colchetes `[]` e a primeira posição de um *array* será sempre 0 (zero). No exemplo abaixo, criamos um *array* chamado `alunos` e inserimos valores nas posições `0`, `1` e `2`.

```
alunos = ['André', 'Pedro', 'Carolina']
```

Como em um *array* cada elemento tem uma posição numerada, podemos recuperar os alunos da seguinte forma:

```
alunos = ['André', 'Pedro', 'Carolina']
alunos[0] # "André"
alunos[1] # "Pedro"
alunos[-1] # último elemento do array, nesse caso, "Carolina"
```

Em algumas linguagens, como Java e C#, precisamos fixar o tipo dos itens de um *array*. Por exemplo: um *array* de *strings* será sempre um *array* de *strings*. Se tentarmos inserir um número ou um *boolean* teremos problemas. Em Ruby, podemos atribuir qualquer tipo de valor dentro de um *array*:

```
alunos = ['André', 1, true]
```

Outra diferença de *arrays* em Ruby é o tamanho da estrutura, ou seja, a quantidade de elementos que ele pode armazenar. Em algumas linguagens, o tamanho do *array* é determinado na sua criação. Em Ruby, podemos adicionar e remover itens sem essa preocupação. Acrescentar ou remover itens de um *array* também é muito simples em Ruby:

```
alunos << 'Laura' # adiciona o valor "Laura" na última posição do
array alunos
alunos.pop() # remove o último valor do array, nesse caso, “Laura”
```

Lembrando das boas práticas de nomenclatura, sempre que declaramos um *array* estamos falando de uma lista/coleção de itens, por isso os nomes de variável serão sempre no plural.

Se você quiser alterar o valor de algum elemento numa posição específica, você pode usar:

```
alunos = ['André', 'Pedro', 'Carolina']
alunos[1] = 'Joaquim' # muda o valor da posição 1 do array de "Pedro"
para "Joaquim"
```

Assim como nas variáveis, em Ruby temos métodos que auxiliam nosso trabalho como:

```
alunos.first() # retorna o valor da primeira posição do array
alunos.last() # retornar o valor da última posição do array
alunos.length() # retorna a quantidade de elementos do array
```

Exercício

Vamos criar duas listas, uma com os nomes de alunos e outra com suas notas. Crie dois *arrays* `alunos = ['André', 'Sophia', 'Laura']` e `notas = [5, 6, 8]`. Para esse exercício, mantenha fixo o código em que os *arrays* são criados. Utilize os comandos para manipulação de *arrays* para executar as orientações a seguir.

Imprima os nome de cada aluno seguido da sua nota. O resultado deve ficar assim:

```
André tirou nota 5
Sophia tirou nota 6
Laura tirou nota 8
```

Imagine que a nota de Sophia está errada, então agora é necessário corrigir o valor para 9. Além disso, precisamos incluir o aluno Paulo, que tirou nota 7.5, na criação dos *arrays*. Adicione nos final das listas os valores em seus respectivos *arrays* e, em seguida, imprima novamente os alunos e suas notas.

```
André tirou nota 5
Sophia tirou nota 9
Laura tirou nota 8
Paulo tirou nota 7.5
```

Agora que temos uma lista de alunos podemos usar o método `length` para imprimir a quantidade de alunos nessa turma. Imprima a frase `“Essa turma possui x participantes”` em que `x` deve ser o valor retornado com `alunos.length`.

Praticamos algumas maneiras de manipular *arrays* e lidar com diferentes tipos de dados. Agora, vamos aprender um pouco sobre dois tipos de dados um pouco mais complexos, mas que nos ajudam muito a lidar com diferentes tipos de informação.

Hash e *Symbol*

Em Ruby temos o *hash* e o *symbol* que são tipos de dados muito utilizados, mas que não estão disponíveis em todas as linguagens.

Symbol

Symbols são muito semelhantes às *strings*, mas ao invés de usar aspas, colocamos `:` para defini-los.

```
“caneca” # string
:caneca  # symbol
```

A grande diferença entre *symbol* e *string* é a forma como o Ruby armazena esses tipos de dados em memória. Toda vez que criamos uma *string*, um novo espaço de memória é alocado para armazená-la, mesmo que seja declarada várias vezes a mesma *string*. Para ver um exemplo disso, vamos usar no IRB um método chamado `object_id()` que traz o número identificador de cada objeto.

```
"caneca".object_id()
 => 70127370789760

"caneca".object_id()
 => 70127366966200

"caneca".object_id()
 => 70127366984720
```

Apesar de repetirmos o valor da *string* três vezes, cada uma delas foi armazenada separadamente. Os *symbols* são declarados de forma similar a uma *string* no código, mas após a primeira declaração o mesmo objeto é reaproveitado. Repare que o `object_id` se mantém nas três chamadas abaixo.

```
:caneca.object_id()
 => 1524188

:caneca.object_id()
 => 1524188

:caneca.object_id()
 => 1524188
```

Os *symbols* são normalmente usados quando precisamos de identificadores. Em Ruby, quando é feita uma comparação entre *strings*, ele precisa verificar cada caractere para saber se há igualdade. Já no caso de *symbols*, basta fazer uma comparação entre números inteiros (`object_id`), o que é muito mais rápido e eficiente.

Hashes

Até agora usamos coleções de elementos simples como: *string*, *integer* e *boolean*. Se queremos guardar diferentes tipos de dados de uma pessoa, por exemplo, teremos algo assim:

```
aluno = ['João', 7, 'Ciências']
```

Para recuperar os valores de um *array* usamos a posição de cada item, certo? Então podemos obter o nome e a disciplina do aluno com o código abaixo:

```
aluno[0]
 => "João"
aluno[2]
 => "Ciências"
```

Um programador com olhos treinados consegue perceber o padrão, mas, mesmo assim, essa estrutura não é boa para a organização do nosso código. Imagine várias linhas ao longo do código onde você precisa recuperar dados de um aluno, o código vai ficar muito difícil de ler e entender.

Aí entra uma estrutura muito importante em Ruby, o *hash*, que é um conjunto de pares com uma **chave** e um **valor**. Com ele, podemos nomear nossos atributos, como um dicionário.

```
aluno = { nome: 'João', nota: 7, disciplina: 'Ciências' }
puts aluno[:nome]
puts aluno[:disciplina]
aluno[:nome] = 'Maria'
puts aluno[:nome]
```

Para criar um *hash* em Ruby basta usar chaves `{}` e, dentro dessas chaves, declarar os pares de chave e valor separados por `:`. Para acessar um dado de um *hash*, basta usar colchetes `[]` com a chave correspondente. O comando retornará o valor associado à chave. Caso a chave não exista, será retornado `nil` (nulo).

Note que nas chaves do *hash* usam *symbols*. Isso não é obrigatório, mas é muito frequente para tirar proveito da melhor performance de comparação dos *symbols*.

Exercício

Utilizando a estrutura de *hash*, crie variáveis de alunos que possuem os seguintes atributos: `nome`, `nota` e `disciplina`. Por exemplo: `alberto = { nome: 'Alberto', nota: 7, disciplina: 'Artes' }`. Em seguida, crie um *array* de alunos chamado alunos e o popule com os `alunos` criados.

Utilize o que você aprendeu até agora para resgatar do *array* e imprimir na tela o nome do aluno, seguido da sua nota e a disciplina. No exemplo anterior o texto impresso ficaria assim:

```
Alberto tirou nota 7 em Artes
```

Dica

Você pode encadear chamadas de *arrays* e *hashes*:

```
alunos = [{ nome: 'Alberto', nota: 7, disciplina:
'Artes' }, { nome: 'Ingrid', nota: 10, disciplina:
'Biologia' }]
# Posição 0 do Array chave nome do Hash
alunos[0][:nome]
```

Estruturas de controle condicional

Nossos exemplos até agora executaram ações linearmente, mas muitas vezes queremos tomar decisões e mostrar resultados relativos a uma condição! Por exemplo,

vimos métodos para verificar se o número é par ou ímpar e podemos querer tomar decisões baseadas nisso:

```
7.odd?() # verifica se 7 é ímpar e retorna true
4.even?() # verifica se 4 é par e retorna true
```

O principal uso desse tipo de método é construir operações de condição. Para criar uma condição em Ruby, usamos a palavra `if` seguida da condição que queremos testar, por exemplo:

```
if 1.odd?()
  puts '1 é um número ímpar.'
end
```

A primeira linha do código acima é uma condição “se 1 for par”. Na linha seguinte, com um recuo de dois espaços para mostrar que o código está dentro do `if`, há um código que só será executado se a condição for verdadeira. Então terminamos com a palavra `end`. Ela mostra que acabamos de escrever tudo o que queremos executar se aquela condição for verdadeira.

Dica

Muitas linguagens utilizam 4 espaços ou uma tabulação (tab) para esse processo de mostrar aninhamento. A convenção da comunidade Ruby é de utilizar dois espaços.

Trazendo para um exemplo mais prático, podemos comparar valores, por exemplo:

```
numero = 1
outro_numero = 20

if numero == outro_numero
  puts "Os números #{numero} e #{outro_numero} foram testados"
  puts 'Os números são iguais'
end
```

Quando usamos `==` estamos comparando valores e verificando se são iguais. Podemos também usar os outros operadores citados anteriormente para realizar comparações.

O parceiro de crime do `if` é o `else` que é executado se a condição do `if` for falsa. Por exemplo:

```
numero = 1
outro_numero = 20

if numero == outro_numero
  puts 'Os números são iguais'
else
  puts 'Os números são diferentes'
end
```

Exercício

Utilizando o *array* de alunos criado no exercício anterior, crie uma estrutura de condição para cada aluno que avalie se sua nota foi inferior a 5 e imprima a situação. Considerando o exemplo da atividade anterior, se o aluno estivesse aprovado (nota maior ou igual a 5), a seguinte frase deveria ser impressa: “Alberto foi aprovado(a) em Artes”. Se estivesse reprovado, seria impressa a frase “Alberto foi reprovado(a) em Artes”.

Switch/Case

Uma outra maneira de utilizarmos condições no nosso código consiste na estrutura chamada de *switch*. Com ela podemos facilmente definir uma série de condições e respostas, reduzindo muito a quantidade de código escrita. A estrutura *switch* consiste em definir a variável que será avaliada, seguida das condições e respostas, por exemplo:

```
nota = 7
case nota
when 0
  puts 'Você tirou zero! Precisa melhorar…'
when 1..4
  puts 'Reprovado… precisa se esforçar mais...'
when 5
  puts 'Passou raspando!'
when 6..9
  puts 'Parabéns, você foi aprovado.'
else
  puts 'Tirou 10! Você deve ser o melhor aluno que já tive!'
end
```

Se você rodar o código acima, pode fazer testes alterando o valor de `nota` e observar o comportamento da aplicação. A linha `case nota` determina o nome da variável que o *switch* vai avaliar. As linhas que começam com `when` apresentam as diferentes condições que o código deve usar para comparar com `nota`. Quando utilizamos a notação `1..4`, por exemplo, estamos orientando que `nota` seja comparada a todos os valores entre 1 e 4.

Exercício

Note que, no exemplo acima, se atribuímos um valor negativo ou maior do que 10 à `nota`, o *switch* imprime uma resposta incoerente, afinal não é possível tirar uma nota negativa ou maior que 10. Modifique seu código para corrigir esses problemas.

Estruturas de repetição

Nos exercícios anteriores vimos como criar uma *hash* que armazena os nomes dos alunos, uma disciplina e uma nota, e executar ações de acordo com condições predeterminadas. No entanto, sempre que queremos imprimir alguma informação na tela, repetimos a mesma linha de comando para cada aluno, o que é ineficiente e gera códigos longos.

Para nos ajudar nessa tarefa as linguagens de programação possuem as estruturas de repetição.

While

O `while` é uma estrutura de repetição que reproduz um trecho de código enquanto uma determinada condição é satisfeita. Veja o exemplo a seguir:

```
tecla_pressionada = 'n'
while tecla_pressionada != 's' do
  puts 'Vou continuar imprimindo até você apertar s'

  tecla_pressionada = gets().chomp() #chomp come o enter do final do
gets
end
```

No código acima nosso programa vai imprimir a frase 'Vou continuar imprimindo até você apertar s' até a **tecla s** ser pressionada.

Dica

O `while` sempre fica rodando até ele ter uma condição de saída, mas se quisermos **abortar** o programa em Ruby podemos usar o atalho `Ctrl + c` no Terminal.

Quando estamos usando `while`, **sempre** temos que garantir uma condição de saída, para o nosso *loop* não ficar rodando para sempre.

```
condicao = 1
while condicao == 1 do
  puts 'A condição está sendo satisfeita'
end
# Lembre-se que pode abortar a execução com Ctrl + c
```

A situação apresentada acima não resulta em erro, mas o computador ficará infinitamente rodando o código dentro do `while`, afinal, a condição sempre será satisfeita. É importante garantir que podemos sair do `while`, senão acabamos por criar o famoso ***Loop* Infinito**. Para garantir que temos uma forma de sair, podemos criar uma nova opção de saída e utilizá-la no *loop*:

```
condicao = 1
while condicao == 1 do
  puts 'A condição está sendo satisfeita'
  condicao = 2
end
```

Dessa maneira simples, modificamos o estado da condição para sair da estrutura de repetição. Assim, a frase será impressa apenas uma vez e sairá do *loop*. Agora vamos dar continuidade ao nosso exemplo e utilizar a estrutura de repetição para inserir alunos num *array* até terminarmos a lista.

```
tecla_pressionada = 's'
alunos = []
while tecla_pressionada == 's' do
  puts 'Digite o nome do aluno: '
  nome_aluno = gets.chomp
  alunos << nome_aluno

  puts 'Deseja inserir um novo aluno? s ou n'
  tecla_pressionada = gets.chomp
end
```

Exercício

Você consegue compreender como funciona o código acima? Modifique-o para armazenar, além do nome do aluno, a sua nota e a disciplina.

For

O `for` é uma estrutura de repetição finita onde sabemos quantas vezes queremos executar uma ação. Sabemos que a primeira posição de um *array* é o 0, seu último elemento é seu tamanho menos um (`alunos.length() - 1`, por exemplo), assim podemos construir uma estrutura de `for` que rode um determinado trecho de código para cada elemento dentro do *array*.

Para fins de comparação com outras linguagens, abaixo temos um exemplo de como fazer um `for` em C#. Note que dentro da estrutura do `for` uma *integer* `i` é declarada e a ela é atribuído o valor 0, depois definimos a condição que `i` deve obedecer para que o código seja rodado (`i` deve ser menor que a quantidade de itens no *array*), em seguida somamos 1 a `i`, até que a condição seja satisfeita e o *loop* seja encerrado.

```
for (int i = 0; i < um_array.length() - 1; i++) {
  print(um_array[i]);
}
```

Dessa maneira, para cada vez que o código rodar, `i` vai aumentar seu valor em 1 e um texto será impresso na tela.

Em Ruby essa estrutura existe, mas não é muito comum usarmos esse tipo de iterador. Temos muitos métodos que realizam a mesma função de forma mais legível. Para percorrer itens de um *array*, por exemplo, temos a opção de usar o método `each`. Ao acionar esse método devemos definir uma variável que vai representar cada item do *array* durante o *loop*. No exemplo abaixo a variável que usamos é `um_aluno`.

```
alunos.each do |um_aluno|
  puts(um_aluno[:nome])
end
```

Com o uso do `each` vimos um novo elemento do Ruby em ação: os blocos. Todo código presente entre o `do` e o `end` representa um bloco e esse bloco é usado como parâmetro para o `each`. De forma simples e prática: se temos 10 itens no *array* de alunos, o código do bloco vai ser executado 10 vezes e a cada execução teremos um dos 10 itens na variável `um_aluno`.

Métodos

Já falamos que métodos são processos que executam determinadas ações nos elementos nos quais são aplicados. Já estamos usando métodos em todos os tipos de dados que apresentamos, em *strings* você usou o método `length()` e `downcase()`, em *arrays* usou o método `pop()`. No entanto, nós podemos construir métodos personalizados no nosso código para executarem funções importantes dentro do nosso programa. Assim,

dizemos que métodos são utilizados para encapsular pequenos comportamentos de código que queremos reaproveitar e executar diversas vezes.

Cada linguagem de programação traz, em seus tipos de dados, métodos diferentes. Ruby é famosa por trazer muitos métodos prontos para facilitar a vida do programador. Em outras linguagens você pode precisar escrever métodos para, por exemplo, descobrir se um número é par ou ímpar.

Em Ruby, métodos são declarados da seguinte maneira:

```
def nome_do_metodo() #os parênteses são opcionais
  puts 'Que método legal!'
  # mais linhas de código
end
```

Assim, quando chamarmos o método no nosso código, essas linhas serão executadas somente naquele momento.

```
# declaração do método
def nome_do_metodo() #os parênteses são opcionais
  puts 'Que método legal!'
  # mais linhas de código
end

# chamada
nome_do_metodo() # método é executado neste momento
# imprime na tela a texto “Que método legal!”
```

Dica

Lembre-se que a definição do método tem que acontecer antes do seu uso!

Métodos com parâmetro e retorno

Em Ruby (e quase todas as outras linguagens), os métodos podem ter parâmetros. Isso quer dizer que você pode passar variáveis, valores ou objetos para o método executar e processar internamente.

No exemplo abaixo, declaramos um método que recebe dois valores e depois irá fazer a soma.

```
def soma_valores(valor1, valor2) #os parênteses são opcionais
  return valor1 + valor2
end
```

Quando vamos chamar este método, nós temos que dizer quais os valores que queremos somar:

```
soma_valores(10, 20) #somamos 10 com 20, retorna 30
```

Agora, diferente das outras linguagens, os métodos em Ruby sempre retornam alguma coisa, mesmo que seja `nil` (nulo). Esse retorno não precisa ser explícito com a palavra `return` como no exemplo, já que a última operação é retornada automaticamente.

```
def soma_valores(valor1, valor2) #os parênteses são opcionais
  valor1 + valor2
end

soma = soma_valores(10, 20) #última operação do método foi a soma
# => 30
```

Vamos utilizar nosso exemplo desenvolvido aqui e criar um método para imprimir os nomes dos alunos com suas notas e as disciplinas. Vamos definir um método chamado `imprime_alunos` que recebe como parâmetro um nome, uma nota e uma disciplina.

```
def imprime_alunos(nome, nota, disciplina)
  puts nome + " tirou nota " + nota + " em " + disciplina
end
```

Agora podemos utilizar esse método no nosso *loop* `alunos.each` para imprimir as informações na tela.

```
alberto = { nome: 'Alberto', nota: 7, disciplina: 'Artes' }
joana = { nome: 'Joana', nota: 8, disciplina: 'Bio' }
karen = { nome: 'Karen', nota: 9, disciplina: 'Filosofia' }
alunos = [alberto, joana, karen]

def imprime_alunos(nome, nota, disciplina)
  puts "#{nome} tirou nota #{nota} em #{disciplina}”
end

alunos.each do |aluno|
  imprime_alunos(aluno[:nome], aluno[:nota], aluno[:disciplina])
end
```

Que tal agora você criar um método que verifica a nota do aluno e retorna se foi aprovado ou não na matéria?

Terminamos aqui o conteúdo sobre Lógica de Programação que queríamos passar. Esse material está em constante evolução e sua opinião é muito importante. Se tiver sugestões ou dúvidas que gostaria de nos enviar, entre em contato pelo nosso e-mail: qsd@campuscode.com.br.

Versão	Data de atualização
1.0.6	20/06/2022