Learn Ruby programming from 0

Let TipsMake.com learn about how to learn Ruby programming language from 0 in this article!

Let TipsMake.com learn about how to learn Ruby programming language from 0 in this article!

1. Guide the most simple and effective way to write easy-to-read code
2. Why should you learn Python programming language?
3. 12 extremely useful tricks for JavaScript programmers

Why should you learn Ruby programming language?

For me - the author of the article, first Ruby is a beautiful language. You will find that writing code in Ruby is smooth and natural.

Secondly, one of the reasons Ruby is popular lies in Rails: a framework used by Twitter, Basecamp, Airbnb, Github and many others.

' *Ruby has a simple appearance, but the inner soul is as complex as a human* .' - **Matz** , the father of Ruby programming language.

Introduction / Origin

Ruby is an extremely powerful open source programming language with simplicity and efficiency. With elegant syntax makes reading and writing extremely easy.

1. Ruby programming language for beginners

Let's start with the basics of Ruby's platform!

Variables (Variables)

You can understand simply that the variable is a word that contains a value. Just simple as that!

In Ruby, it's easy to define a variable and give it a value. Suppose you want to give a value of 1 to a variable called `one` . Do it:

```
one = 1
```

Is that simple? You only need to assign a value of 1 to a variable called `one` .

```
two = 2
some_number = 10000
```

You can assign any value to any variable you want. In the above example, variable `two` contains an integer value of 2 and `some_number` has a value of 10,000.

Besides integers, we can also use booleans (true / false), string, symbol, float and other data types.

```
 booleans
true_boolean = true
false_boolean = false

# string
my_name = "Leandro Tk"

# symbol
a_symbol = :my_symbol

# float
book_price = 15.80
```

Conditional statement: Control line

Conditional statements will be responsible for evaluating the correctness / wrongness of a command. If the result is correct (True), the content will be processed. For example:

```
if true
 puts "Hello Ruby If"
end

if 2 > 1
 puts "2 is greater than 1"
end
```

2 is greater than 1 so the text ' *2 is greater than 1* ' will be displayed.

The `else` will be executed when we get False (False):

```
if 1 > 2
 puts "1 is greater than 2"
elsif 2 > 1
 puts "1 is not greater than 2"
else
 puts "1 is equal to 2"
end
```

1 is not greater than 2 so the code in the `else` command will be processed.

In addition, you can also use the `elsif` command, which combines both. For example:

```
if 1 > 2
 puts "1 is greater than 2"
elsif 2 > 1
 puts "1 is not greater than 2"
else
 puts "1 is equal to 2"
end
```

I like to use the `if` after the code has been executed:

```
def hey_ho?
 true
end

puts "let's go" if hey_ho?
```

Looks very beautiful but short. That is the strength of Ruby programming language.

Looping / Iterator (Loop)

In the Ruby programming language, we can repeat in many different forms. I will talk about three loops: **while** , **for** and **each** .

1. **While loop** : As long as the result of statement is still true, the code content contained in statement will always be executed. Thus, the code will print and display from 1 to 10 as in the example below:

```
num = 1

while num = 10
 puts num
 num += 1
end
```

1. **For loop** : With the statement given, the code content will be executed until the request of that statement is satisfied. You pass the num variable to the block and the `for` statement will repeat it for you. This code will print like when the code is between 1 and 10:

```
for num in 1.10
 puts num
end
```

1. **Each loop** : I really like the `each` loop. For an array of values, it will repeat each value.

```
[1, 2, 3, 4, 5].each do |num|
 puts num
end
```

The difference between **For** and **Each** is `Each` performs exactly the given value; while `For` will appear unexpected values ??outside the request.

```
# for vs each

# for looping
for num in 1.5
 puts num
end

puts num # => 5

# each iterator
[1, 2, 3, 4, 5].each do |num|
 puts num
end


puts num # => undefined local variable or method `n' for main:Object (NameError)
```

Array (Array): Collection / List / Data Structure

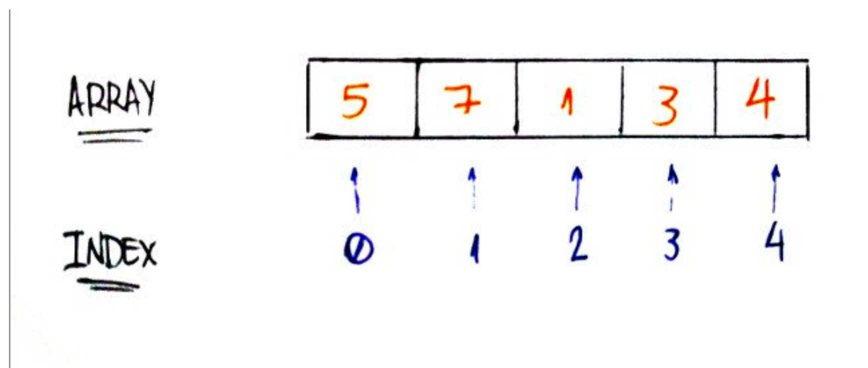
Suppose you want to store an integer 1 into a variable. But now, you want to save 2 and try 3,4,5 . So is there a way to save the integers you want without having to do it manually? Ruby will provide a solution for you.

Array (array) is a collection used to store a list of values ??(like these integers). So use it.

```
my_integers = [1, 2, 3, 4, 5]
```

Very simply, we create an array and save it to **my_integer** .

You will ask yourself: ' *How do I get values ??from that array?* . In Arrays there is a concept called **Index** . Start with index 0 and keep increasing.



Using Ruby syntax is simple to understand:

```
my_integers = [5, 7, 1, 3, 4]
print my_integers[0] # 5
print my_integers[1] # 7
print my_integers[4] # 4
```

Now if you want to save strings instead of integers, like your relative name list, for example:

```
relatives_names = [
 "Toshiaki",
 "Juliana",
 "Yuji",
 "Bruno",
 "Kaio"
]

print relatives_names[4] # Kaio
```

Nothing is different, except now we use words instead of numbers. Great!

We have learned how the array index works. Now add the elements to the array data structure (items on the list).

The usual method to add values ??to an array is push and .

Extremely simple push. You only need to pass the elements (*The Effective Engineer*) as a push parameter:

```
bookshelf = []
bookshelf.push("The Effective Engineer")
bookshelf.push("The 4 hours work week")
print bookshelf[0] # The Effective Engineer
print bookshelf[1] # The 4 hours work week
```

Method is a little different:

```
bookshelf = []
bookshelf  "Lean Startup"
bookshelf  "Zero to One"
print bookshelf[0] # Lean Startup
print bookshelf[1] # Zero to One
```

You can ask the question: " *Why doesn't it use dot notation like other methods? How can it become a method?* '. A good question. Written as follows. :

```
bookshelf  "Hooked"
```

Is also.

```
bookshelf.("Hooked")
```

Ruby programming language is great, isn't it? Now let's talk about another type of Data Structure.

Hash: Key-Value Data Structure / Dictionary Collection

We all know arrays are essentially arrays with numbers. But what if we use the serial number? Some data structures can use numbers, strings or other types of data. The Hash data structure is one of them.

Hash is a collection of key-value pairs:

```
hash_example = {
 "key1" => "value1",
 "key2" => "value2",
 "key3" => "value3"
}
```

In it, the key will refer to the index of the value. So how will we access the value of Hash? Will pass the key.

This is a hash about me using the key as the name, nickname and ethnicity:

```
hash_tk = {
 "name" => "Leandro",
 "nickname" => "Tk",
 "nationality" => "Brazilian"
}

print "My name is #{hash_tk["name"]}" # My name is Leandro

print "But you can call me #{hash_tk["nickname"]}" # But you can call me Tk


print "And by the way I'm #{hash_tk["nationality"]}" # And by the way I'm Brazil
```

In the above example, I give a phrase about myself using all the values ??stored in the hash.

One pretty cool thing about a hash is that we can use anything as a value. I will add the 'age' key and my age number is (24) to the value.

```
hash_tk = {
 "name" => "Leandro",
 "nickname" => "Tk",
 "nationality" => "Brazilian",
 "age" => 24
```

```
}

print "My name is #{hash_tk["name"]}" # My name is Leandro

print "But you can call me #{hash_tk["nickname"]}" # But you can call me Tk

print "And by the way I'm #{hash_tk["age"]} and #{hash_tk["nationality"]}" # And
```

Now we will add other elements to a hash. Key leads to value is the characteristic of making Hash so the addition will follow the same rules.

```
hash_tk = {
  "name" => "Leandro",
  "nickname" => "Tk",
  "nationality" => "Brazilian"
}

hash_tk["age"] = 24

print hash_tk # { "name" => "Leandro", "nickname" => "Tk", "nationality" => "Bra
```

We just need to assign a value to the hash. Do you find it simple to add a value to the hash?

Iteration: Loop through Data Structures

Repeating arrays is very simple. Ruby developers often use `each` loop. Let's do it:

```
bookshelf = [
  "The Effective Engineer",
  "The 4 hours work week",
  "Zero to One",
  "Lean Startup",
  "Hooked"
]

bookshelf.each do |book|
  puts book
end
```

The `each` loop above works by passing each element in the array as a parameter in the block. In the above example, we will print each of those elements.

With the hash data structure, we can also use `each` loop to go through two parameters in the same block: key and value. Here is an example:

```
hash = { "some_key" => "some_value" }
hash.each { |key, value| puts "#{key}: #{value}" } # some_key: some_value
```

We will name these two parameters as the key and the value from being mistaken.

```
hash_tk = {
 "name" => "Leandro",
 "nickname" => "Tk",
 "nationality" => "Brazilian",
 "age" => 24
}

hash_tk.each do |attribute, value|
 puts "#{attribute}: #{value}"
end
```

You can see that we used the property as a parameter for the hash key and it worked. Great!

Classes & Objects

As an object oriented programming language, Ruby uses concepts of class and object.

"Class" is a way to identify objects. In the real world there are many objects of the same type as vehicles, dogs and bicycles. Each car has the same components (wheels, doors, engines).

'Objects' has two characteristics: data and behavior. For example, cars have data on wheel numbers and door numbers. They also act like accelerating and stopping.

In object-oriented programming, we call " *attributes* " data and " *methods* " behavior.

Data = Attributes
Behavior = Methods

Object-oriented programming mode in Ruby: On (On)

This is a syntax in Ruby for Class:

```
class Vehicle
end
```

We specify `Vehicle` with the class command and end with end. So easy!

Objects are representative of the class. We create an instance using the **.new** method.

```
vehicle = Vehicle.new
```

Here, the `vehicle` is an object (or instance) of class: Vehicle

Class `Vehicle` has 4 attributes: wheel, tank type, seat number and speed.

Identify our Vehicle class to receive data and create specific objects.

```
class Vehicle

def initialize(number_of_wheels, type_of_tank, seating_capacity, maximum_veloci
```

```
  @number_of_wheels = number_of_wheels
  @type_of_tank = type_of_tank
  @seating_capacity = seating_capacity
  @maximum_velocity = maximum_velocity
  end
 end
```

We used the initialize method. Another name is the constructor because when you create the vehicle object, it also determines its attributes.

Suppose you like the Tesla Model S and want to create such an object. Includes 4 wheels, electric car, 5 seats and running at maximum speed of 250km / hour (155 mph). Create the tesla_model_s object!

```
 tesla_model_s = Vehicle.new(4, 'electric', 5, 250)
```

4 wheels + electric tank + 5 seats + 250km / hour maximum speed = tesla_model_s.

(4 wheels + electric car + 5 seats + maximum speed 250km / hour = tesla_model_s.)

```
 tesla_model_s
 # =>
```

So we have set up the Tesla attribute. But how to access?

We send a message to the subject to ask about them. We call that a method. That is the behavior of the object. Let's do it!

```
 class Vehicle

 def initialize(number_of_wheels, type_of_tank, seating_capacity, maximum_velocit

  @number_of_wheels = number_of_wheels
  @type_of_tank = type_of_tank
  @seating_capacity = seating_capacity
  @maximum_velocity = maximum_velocity
  end

  def number_of_wheels
  @number_of_wheels
  end

  def set_number_of_wheels=(number)
  @number_of_wheels = number
  end
 End
```

In the above example, we use two ways of number_of_wheels and set_number_of_wheels. Also known as getter and setter. First, we take an attribute value and then, set a value for the attribute.

In Ruby, we can do that without using the above methods with attr_reader, attr_writer and attr_accessor.

1. **attr_reader** : Apply the getter method.

```
class Vehicle
 attr_reader :number_of_wheels


def initialize(number_of_wheels, type_of_tank, seating_capacity, maximum_velocit

 @number_of_wheels = number_of_wheels
 @type_of_tank = type_of_tank
 @seating_capacity = seating_capacity
 @maximum_velocity = maximum_velocity
 end
end

tesla_model_s = Vehicle.new(4, 'electric', 5, 250)
tesla_model_s.number_of_wheels # => 4
```

1. **attr_writer** : Apply setter method.

```
class Vehicle
 attr_writer :number_of_wheels


def initialize(number_of_wheels, type_of_tank, seating_capacity, maximum_velocit

 @number_of_wheels = number_of_wheels
 @type_of_tank = type_of_tank
 @seating_capacity = seating_capacity
 @maximum_velocity = maximum_velocity
 end
end

# number_of_wheels equals 4
tesla_model_s = Vehicle.new(4, 'electric', 5, 250)
tesla_model_s # =>

# number_of_wheels equals 3
tesla_model_s.number_of_wheels = 3
tesla_model_s # =>
```

1. **attr_accessor** : Apply both methods.

```
class Vehicle
 attr_accessor :number_of_wheels


def initialize(number_of_wheels, type_of_tank, seating_capacity, maximum_velocit

 @number_of_wheels = number_of_wheels
 @type_of_tank = type_of_tank
 @seating_capacity = seating_capacity
 @maximum_velocity = maximum_velocity
 end
end

# number_of_wheels equals 4
```

```
tesla_model_s = Vehicle.new(4, 'electric', 5, 250)
tesla_model_s.number_of_wheels # => 4

# number_of_wheels equals 3
tesla_model_s.number_of_wheels = 3
tesla_model_s.number_of_wheels # => 3
```

Now we have learned how to get attribute values; implement getter and setter methods; and use attr (reader, writer and accessor).

Besides, we can also use methods to do other things - like the "make_noise" method. Try it!

```
class Vehicle

def initialize(number_of_wheels, type_of_tank, seating_capacity, maximum_velocit

 @number_of_wheels = number_of_wheels
 @type_of_tank = type_of_tank
 @seating_capacity = seating_capacity
 @maximum_velocity = maximum_velocity
 end

 def make_noise
 "VRRRRUUUUM"
 end
End
```

When we call this method, it only returns a string "VRRRRUUUUM".

```
v = Vehicle.new(4, 'gasoline', 5, 180)
v.make_noise # => "VRRRRUUUUM"
```

Encapsulation: Hide information

Encapsulation is a way of limiting the direct access to the object's data and methods and facilitating the operation on that data (the object's method).

In object-oriented programming, inside each object contains data that represents its status or properties. Each object is equipped with behavior or method to perform certain tasks, to inform or change its own properties. The object is a combination of data and manipulation on that data into a unified whole. This combination is called encapsulation.

In other words, the inside information will be hidden, only the new object can interact with the internal data.

Thus, in the Ruby programming language, we will use a method to access data directly. Take an example:

```
class Person
 def initialize(name, age)
 @name = name
 @age = age
 end
```

```
end
```

Thus, we have applied the Person class.

We implement this Person class. As we learned, to create the person object, we use the new method and pass the parameters.

```
tk = Person.new("Leandro Tk", 24)
```

So what to do to access this information?

```
tk.name
> NoMethodError: undefined method `name' for #
```

It is impossible to do! We did not implement the name method (and age).

Remember: " *In Ruby, what methods do we use to directly access data?* " To access the name and age `tk` we need to implement methods on our Person class.

We can now access this information directly. With encapsulation, we can ensure that the object (tk in this case) is only allowed to access the name and age. The internal representation of the object is hidden from the outside.

Obviously not. The reason because we have to use the direct access method as follows:

```
class Person
 def initialize(name, age)
 @name = name
 @age = age
 end

 def name
 @name
 end

 def age
 @age
 end
end
```

Inheritance: Behavior and Characteristics

Certain objects have common points: Behavior and characteristics.

For example, I inherited some characteristics and behaviors from my father - like eyes and hair. Along with behaviors like impatience and introvert.

In object-oriented programming, classes can inherit common characteristics (data) and behavior (methods) from another class.

Let's look at another example and do it in Ruby.

Imagine a car. The maximum number of wheels, seats and velocities are all properties of a vehicle.

```
class Car
 attr_accessor :number_of_wheels, :seating_capacity, :maximum_velocity

 def initialize(number_of_wheels, seating_capacity, maximum_velocity)
 @number_of_wheels = number_of_wheels
 @seating_capacity = seating_capacity
 @maximum_velocity = maximum_velocity
 end
end
```

Class Car was made:

```
my_car = Car.new(4, 5, 250)
my_car.number_of_wheels # 4
my_car.seating_capacity # 5
my_car.maximum_velocity # 250
```

In Ruby, we use `operator` to specify that a class inherits from another class. The ElectricCar class will inherit from the Car class.

```
class ElectricCar  Car
end
```

Simply put, we don't need to use any more methods because this class already has information inherited from Car.

```
tesla_model_s = ElectricCar.new(4, 5, 250)
tesla_model_s.number_of_wheels # 4
tesla_model_s.seating_capacity # 5
tesla_model_s.maximum_velocity # 250
```

Great!

Module: Toolbox

We can think of a module as a toolbox containing a set of constants and methods.

An example of Ruby module is Math. We can access the constant PI:

```
Math::PI # > 3.141592653589793
```

And the .sqrt method:

```
Math.sqrt(9) # 3.0
```

In addition, we can apply our module and use it in the class. We have a class RunnerAthlete:

```
class RunnerAthlete
 def initialize(name)
 @name = name
 end
end
```

Apply the Skill module to get the average_speed method.

```
module Skill
 def average_speed
 puts "My average speed is 20mph"
 end
end
```

How to add a module to the class so that it obtains this behavior (average_speed)? Simply write it straight there!

```
class RunnerAthlete
 include Skill

 def initialize(name)
 @name = name
 end
end
```

You can see the 'include skill' section! Now you can use this method in the instance of class RunnerAthlete.

```
mohamed = RunnerAthlete.new("Mohamed Farah")
mohamed.average_speed # "My average speed is 20mph"
```

However, you need to remember the following:

1. modules may not have instances;
2. The module may not have a subclass;
3. Modules are defined by modules.

Congratulations! You have completed the content you need to know about the Ruby programming language! If you have any questions, please let us know in the comment section below!

Author: TK

Refer to some more articles:

1. Arrays and objects in JavaScript are like stories and newspapers!
2. Journey to change jobs from fashion models to software engineers within 1 year
3. If you want to be a data scientist, learn these 3 languages ??right away!

Having fun!

You finished reading the article "**Learn Ruby programming from 0**" edited by the **TipsMake** team. We hope this article has provided you with many useful tech tips and tricks. You can search for similar articles on tips and guides. Thank you for reading and for following us regularly.

© 2019 TipsMake.com